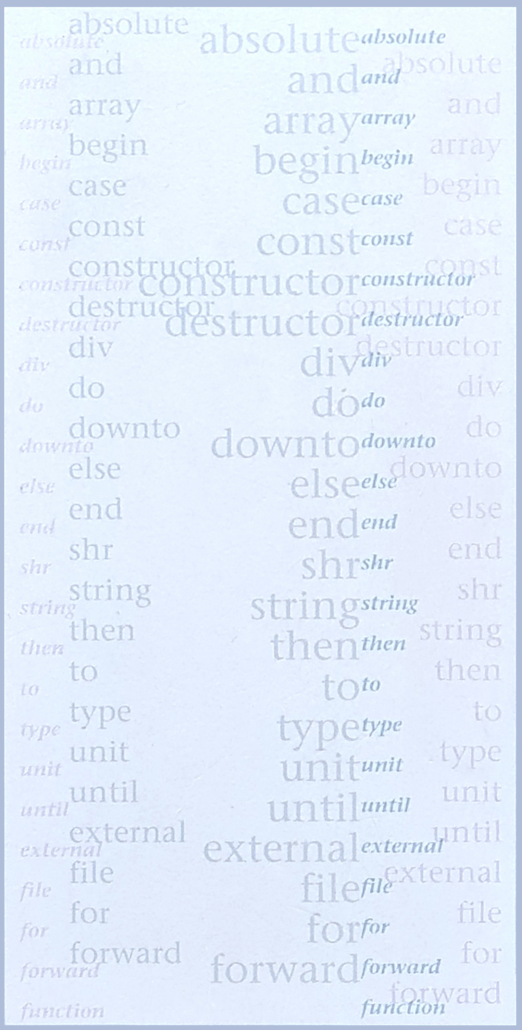
Turbo Pascal 6.0



Referenz-handbuch

Das produktive Pascal-Entwicklungssystem für objektorientierte Programmierung. Komplett mit mausgesteuerter Benutzeroberfläche, integriertem Debugger und Assembler.

B O R L A N D

Turbo Pascal 6.0

Turbo Pascal 6.0

Referenzhandbuch

B O R L A N D

Copyright © 1990 by Borland International, INC.
Jede Vervielfältigung dieses Handbuches sowie des Turbo Pascal Softwareprogrammes wird strafrechtlich verfolgt. Die Rechte an der Dokumentation und die Rechte an dem Softwareprogramm Turbo Pascal liegen bei Borland International.

Der rechtmäßige Erwerb der Programmdisketten und der Handbücher erlaubt die Nutzung der Programme analog der Benutzung eines Buches. Entsprechend der Unmöglichkeit, daß ein Buch zugleich an verschiedenen Orten von mehreren Personen gelesen wird, darf das Softwareprogramm Turbo Pascal nicht gleichzeitig von verschiedenen Personen an verschiedenen Orten und auf verschiedenen Geräten benutzt werden. Diskettenkopien dürfen lediglich zum Zweck der Datensicherung angefertigt werden.

Sie (der Lizenznehmer von Turbo Pascal) dürfen eigene Programme, die Sie mit dem Turbo-Pascal-Sprachcompiler geschrieben und compiliert haben, ohne Einschränkung oder Verpfichtung zu zusätzlichen Lizenzgebühren verwenden, weitergeben oder verkaufen, solange diese Programme einen Copyright-Vermerk enthalten. Unter Copyright-Vermerk verstehen wir entweder Ihren eigenen Copyright-Vermerk oder gegebenenfalls einen Hinweis wie »Created using © Borland International 1990«. Die Originaldisketten von Turbo Pascal beinhalten einige Dateien mit Hardware- und Zeichensatzinformationen, die von dem Standard-Grafik-Unit (GRAPH.TPU) verwendet werden. Diese Dateien mit den Suffixen .CHR und BGI unterliegen besonderen Nutzungsbeschränkungen. Sie dürfen nicht als Dateien weiterverkauft werden. Sie dürfen diese Dateien aber in Ihren eigenen mit Turbo Pascal geschriebenen und compilierten Programmen verwenden. Sie dürfen diese compilierten Programme auch weiterverkaufen, solange sie einen Copyright-Vermerk beinhalten.

Die Beispielprogramme von Turbo Pascal, die auf den Originaldisketten mitgeliefert werden, demonstrieren die verschiedenen Eigenschaften von Turbo Pascal. Sie dienen ausschließlich zu Lehrzwecken. Sie dürfen diese Beispielprogramme editieren oder für den eigenen Gebrauch modifizieren. Sie dürfen diese Programme jedoch nicht als Ganzes oder in Teilen in compilierter Form oder als Quelltext weitergeben oder verkaufen. Sie dürfen die Routinen aus den Beispielprogrammen in eigenen Anwendungsprogrammen verwenden, solange die resultierenden Programme nicht einem Bespielprogramm oder einem Teil eines Beispielprogrammes in Erscheinung oder Funktion gleichen.

Einschränkung der Gewährleistung
Es wird keine Garantie für die Richtigkeit des Inhaltes dieses Handbuches übernommen. Da sich Fehler, trotz aller Bemühungen, nie vollständig vermeiden lassen, sind wir für Hinweise jederzeit dankbar.

Herausgeber
Borland GmbH
Lindwurmstraße 88
Postfach 15 03 40
8000 München 2
Telefon 0 89 / 72 01 00
Telefax 0 89 / 77 93 39
Telex 5 21 26 37 mcm d

Umschlag und typografische Konzeption: Rudolf Paulus Gorbach

Grafiken: Sid Gastl, Andreas Höhne

Übertragung ins Deutsche: Dieter Strehle, Kapitel 1: Alois Stockklauser

PostScript-Belichtung: CD Computer & Dokumentations GmbH, Aalen

Druck und Bindung: Schoder Druck GmbH, Gersthofen

1. Auflage, Oktober 1990

Inhalt

Einführung 1
Was in den Kapiteln zu finden ist 1
Typographie 2

1. Verzeichnis der Routinen 3

2. Compiler-Befehle 179
Schalter 180
Parameter 186
Bedingte Compilierung 188
Symbole zur bedingten Compilierung 189

3. Fehlermeldungen und -codes 193
Fehlermeldungen des Compilers 193
Laufzeitfehler 213
Ein-/Ausgabefehler 213
Fehlercodes von DOS 213
Fehler bei der Bearbeitung von Dateien 215
Critical Errors 216
Fehler mit sofortigem Abbruch (Fatal Errors) 217

4. Zeichensatz und Tastaturcodes 221
ASCII-Codes 221
Erweiterte Tastaturcodes 224
Scancodes der Tastatur 225

Index 227

Einführung

Wie der Titel schon sagt, ist dieser Band zum Nachschlagen gedacht. Wir haben uns bemüht, hier alles zusammenstellen, was selbst ein erfahrerener Programmierer immer wieder nachschlagen muß. Sie finden eine alphabetisch geordnete Liste aller Funktionen und Prozeduren, die dem Programmierer zur Verfügung stehen, darüber hinaus die Compilerbefehle einschließlich der Symbole zur bedingten Compilierung, dann auch die Fehlermeldungen des Compilers und die Laufzeit-Fehlermeldungen. Die ASCII-Tabelle und die Liste der Tastaturcodes schließen diesen Band ab.

Was Sie in diesem Band vergebens suchen, ist eine Einführung in Pascal, die Sie im *Benutzerhandbuch* finden. Dort finden Sie auch Hinweise zur Arbeit mit der integrierten Entwicklungsumgebung, zur Bedienung des Editors usw. Im *Programmierhandbuch* finden Sie die Sprachdefinition von Turbo Pascal, eine Besprechung der Standard-Units, des integrierten Assemblers und weiterer interner Details. Der objektorientierten Programmierung und Turbo Vision ist ebenfalls ein eigener Band gewidmet, der auch die gesamte Referenz zu Turbo Vision enthält.

Was in den Kapiteln zu finden ist

Kapitel 1 »Verzeichnis der Routinen« macht den Hauptteil dieses Bandes aus. Hier werden sämtliche Prozeduren und Funktionen der Units *System*, *Crt*, *Dos*, *Overlay* und *Graph* in alphabetischer Reihenfolge detailliert besprochen.

Kapitel 2 »Compiler-Befehle« erläutert die Compiler-Befehle einschließlich der Symbole zur bedingten Compilierung.

Kapitel 3 »Fehlermeldungen und -codes« erläutert sowohl die Fehlermeldungen, die bei der Compilierung eines Programms eventuell ausgegeben werden, als auch die Codes, die beim Abbruch eines Programms auf den aufgetretenen Fehler hinweisen.

Kapitel 4 »Zeichensatz und Tastaturcodes« besteht aus einer tabellarischen Auflistung des IBM-Zeichensatzes und der von der Tastatur zurückgegebenen Codes.

Typographie

Zur Verdeutlichung von Zusammenhängen verwenden wir in diesem Handbuch mehrere unterschiedliche Schriftarten:

`Schreibmaschine` steht für Syntaxdefinitionen, Eingaben des Benutzers, Ausgaben von Beispielprogrammen und wird für Programmlistings verwendet.

`Schreibmaschine fett` kennzeichnet in Listings reservierte Wörter von Turbo Pascal, zum Beispiel **`procedure`**, **`begin`** usw.

Fettschrift wird im laufenden Text benutzt, um reservierte Wörter (wie **begin**, **end**, **object** usw.) hervorzuheben

Kursivschrift kennzeichnen Namen von Variablen und Bezeichnern im laufenden Text. Außerdem dient diese Schriftart zur Hervorhebung neuer Begriffe.

`Tasten` stehen für die Tasten Ihrer Tastatur, zum Beispiel in »Ein Druck auf `Esc` entfernt das Menü wieder vom Bildschirm«.

1. Verzeichnis der Routinen

Die folgenden rund 200 Seiten enthalten eine alphabetische Auflistung der Prozeduren und Funktionen von Turbo Pascal zusammen mit den Routinen der Standard-Units *Dos*, *Crt*, *Overlay* und *Graph*.

Alle Einträge dieses Verzeichnisses haben dieselbe Form – je nach Art der Routine kann allerdings der eine oder andere Unterpunkt fehlen:

Beispiel-Prozedur — Unitname

Funktion	was die Routine bewirkt.
Deklaration	die formale Deklaration, benutzerdefinierte Elemente sind kursiv gesetzt.
Ergebnistyp	der Typ des zurückgelieferten Wertes (nur bei Funktionen).
Erläuterung	generelle Informationen zur Arbeitsweise.
Grenzen	Einschränkungen und Hinweise auf mögliche Probleme.
Querverweis	auf Routinen ähnlicher, verwandter oder entgegengesetzter Arbeitsweise.
Beispiel	Ein kurzes Programm oder Programmfragment, das diese Routine benutzt.

Hinweise: Die mathematischen Routinen der Unit *System* (*Sin*, *Pi*, *Sqrt* usw.) liefern im Modus `{$N-}` Werte des Typs *Real* zurück; bei der Verwendung eines numerischen Coprozessors bzw. des Emulators (also im Modus `{$N+}`) haben sie dagegen den Ergebnistyp *Extended*.

Der Unitname ist nur angegeben, wenn sich die Routine nicht in *System* befindet. Eine Angabe wie *Graph* bedeutet also, daß zur Verwendung der Routine die Unit *Graph* mit **uses** in das Programm aufgenommen werden muß.

Abs-Funktion

Funktion	liefert den absoluten Wert des Arguments zurück.
Deklaration	`Abs(x)`

Ergebnistyp derselbe wie der Typ des übergebenen Arguments.

Erläuterung *x* ist ein Ausdruck mit einem Integer- oder Real-Typ. Das Ergebnis hat denselben Typ und stellt den absoluten Wert von *x* dar.

Beispiel
```
var
  r: Real;
  x: Integer;
begin
  r := Abs(-2.3);    { r := 2.3; }
  x := Abs(-157);    { x := 157; }
end.
```

Addr-Funktion

Funktion liefert die Adresse des angegebenen Objekts zurück.

Deklaration `Addr(x)`

Ergebnistyp *Pointer*

Erläuterung *x* ist eine beliebige Variable oder der Bezeichner einer Prozedur oder Funktion, die nicht als **inline** deklariert ist. Das Ergebnis ist ein Zeiger, der auf *x* zeigt (also die Adresse von *x* enthält). Das Ergebnis von *Addr* ist wie **nil** zuweisungskompatibel zu allen Zeiger-Typen.

Hinweis: Der Adress-Operator @ hat dasselbe Ergebnis wie die Funktion *Addr*.

Querverweis *Ofs*, *Ptr*, *Seg*

Beispiel
```
var
  p : Pointer;
begin
  p := Addr(p);  { p zeigt jetzt auf sich selbst }
end.
```

Append-Prozedur

Funktion öffnet eine existierende Datei für das Anhängen weiterer Daten.

Deklaration `Append(var f:Text)`

Erläuterung *f* ist eine Textdatei-Variable, die zuvor über *Assign* mit einer externen Datei verbunden worden sein muß.

Append öffnet die Datei und setzt die momentane Position innerhalb dieser Datei an ihr Ende, folgende Schreibaktionen hängen also neue Daten an. War die Datei bereits offen, wird sie durch *Append* zuerst geschlossen und dann erneut eröffnet.

Wenn der letzte 128-Byte-Block der Datei ein ^Z (ASCII-Code 26) enthält, setzt *Append* die momentane Position innerhalb der Datei so, daß die nächste Schreibaktion dieses Zeichen überschreibt. Auf diese Weise läßt sich Text auch an Dateien anhängen, die mit ^Z abgeschlossen wurden.

Wenn *f* mit *Assign(f,'')* ein Nullstring als Name zugeordnet wurde, dann bezieht sich diese Datei-Variable nach dem Aufruf von *Append* auf die Standard-Ausgabe (DOS-Handle 1, entspricht *Output*).

In eine mit *Append* eröffnete Datei kann nur geschrieben werden – jeder Aufruf der Form *Eof(f)* liefert *True* als Ergebnis.

Im Modus {$I-}liefert die Funktion *IOResult* den Wert 0 zurück, wenn *Append* fehlerfrei durchgeführt werden konnte, ansonsten liefert sie einen Fehlercode <>0.

Grenzen — neue Dateien werden von *Append* nicht erzeugt – die mit *f* verbundene Datei muß also bereits existieren.

Querverweis — *Assign*, *Close*, *Reset*, *Rewrite*

Beispiel

```
var f: Text;
begin
  Assign(f, 'TEST.TXT');  { Zuordnung des Namens }
  Rewrite(f);     { erzeugt eine neue (leere) Datei }
  Writeln(f, 'Eine Zeile');
  Close(f);
  Append(f);      { eröffnet die Datei erneut }
  Writeln(f, 'Diese Zeile wird angehängt!');
  Close(f);     { Datei enthält jetzt zwei Textzeilen }
end.
```

Arc-Prozedur — Graph

Funktion — zeichnet einen Kreisbogen bzw. Kreisausschnitt.

Deklaration — `Arc(X,Y: Integer; StAngle , EndAngle , Radius: Word)`

Erläuterung — *Arc* zeichnet einen Kreisbogen mit dem durch das Koordinatenpaar (*X,Y*) angegebenen Mittelpunkt. *StAngle* legt den Startpunkt des mit *Radius* gezogenen Kreisbogens fest, *EndAngle* sein Ende. Die Angabe beider Winkel erfolgt in Grad, wobei entgegen dem Uhrzeigersinn gezählt wird (0 Grad = horizontal rechts des Mittelpunkts, 90 Grad = senkrecht über dem Mittelpunkt usw.).

Arc mit einem Startwinkel von 0 und einem Endwinkel von 360 zeichnet einen vollen Kreis.

Hinweis: Jeder Grafiktreiber (.BGI) enthält einen Wert zur Korrektur der Höhen-/Seitenverhältnisse des Bildschirms (*AspectRatio*), der automatisch von *Circle*, *Arc* und *PieSlice* verwendet wird.

Grenzen vor der Anwendung von *Arc* muß ein Grafikmodus gesetzt worden sein.

Querverweis *Circle*, *Ellipse*, *FillEllipse*, *GetArcCoords*, *GetAspectRatio*, *PieSlice*, *Sector SetAspectRatio*

Beispiel

```
program ArcDemo;
uses Graph;
var
  r, GraphMode, GraphDriver: Integer;

begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
  for r := 1 to 5 do
    Arc(100,100,0,120,r*10);   { fünf Ausschnitte mit
                                 verschiedenen Radien }
  Readln;
  CloseGraph;
end.
```

ArcTan-Funktion

Funktion liefert den Arcustangens des Arguments zurück.

Deklaration `ArcTan(x: Real)`

Ergebnistyp *Real*

Erläuterung *x* ist ein Ausdruck mit einem Real-Typ. Das Funktionsergebnis ist der Arcustangens von *x* (also der Wert, der als Argument einer Tangens-Berechnung den Wert *x* ergeben würde). Die zugrundeliegende Einheit ist *Rad*.

Querverweis *Cos*, *Sin*

Beispiel

```
var
  r : Real;
begin
  r := ArcTan(Pi);
end.
```

Assign-Prozedur

Funktion ordnet einer Datei-Variablen eine externe Datei zu.

Deklaration `Assign(var f; Name : string)`

Erläuterung *f* ist eine Datei-Variable (die für eine typisierte, untypisierte oder eine Text-Datei stehen kann), *Name* ist ein String-Ausdruck.

Assign ordnet *f* die durch *Name* angegebene Datei zu. Alle weiteren Operationen mit *f* beziehen sich solange auf diese Datei, bis *f* durch einen weiteren Aufruf von *Assign* eine andere externe Datei zugeordnet wird.

Ein Dateiname besteht aus einem optionalen Laufwerksbezeichner sowie einer optionalen Anzahl von Verzeichnisnamen, die jeweils durch einen umgekehrten Schrägstrich (backslash) voneinander getrennt sind. Den letzten Teil der Angabe bildet der Name der Datei selbst:

```
Laufwerk:\Dir_Name\...\Dir_Name\Dateiname
```

Wenn die Angabe nach *Laufwerk* mit einem umgekehrten Schrägstrich beginnt, wird sie auf das Stammverzeichnis des jeweiligen Laufwerks bezogen, ansonsten auf das momentan gesetzte Verzeichnis. Wenn *Laufwerk* nicht angegeben ist, wird das momentan gesetzte Laufwerk verwendet. *Dir_Name* und *Dateiname* bestehen jeweils aus maximal 8 Zeichen, *Dateiname* kann ein zusätzliches *Suffix* enthalten (drei Zeichen, abgetrennt durch einen Punkt).

Ein vollständiger Dateiname (Laufwerk, Suchweg und Dateiname selbst) darf maximal 79 Zeichen umfassen.

Wenn *f* einen Nullstring darstellt, dann ordnen *Assign* und ein nachfolgendes *Reset* oder *Rewrite* dieser Datei-Variablen die Standard-Datei *Input* bzw. *Output* zu – auf diese Weise sind Umleitungen der Ein-/Ausgabe über DOS möglich.

Grenzen *Assign* darf nie auf eine offene Datei angewendet werden.

Querverweis *Append*, *Close*, *Reset*, *Rewrite*

Beispiel

```
{ Die Ausgabe dieses Programms läßt sich mit DOS
  beliebig auf einen Drucker, in eine Datei
  etc. umleiten }
var
  f : Text;
begin
  Assign(f, '');   { Standard-Datei }
  Rewrite(f);     { setzt f als Output }
  Writeln(f, 'Diese Ausgabe läßt sich umleiten!');
  Close(f);
end.
```

AssignCrt-Prozedur Crt

Funktion ordnet dem Bildschirm (CRT) eine Textdatei zu.

Deklaration `AssignCrt(var f:Text)`

Erläuterung *AssignCrt* arbeitet exakt so wie die Standardprozedur *Assign*. Der einzige Unterschied liegt darin, daß kein Dateiname ange-

geben wird: *AssignCrt* verbindet die Datei-Variable automatisch mit dem Bildschirm.

Ein- und Ausgabefunktionen gehen über *Crt* mit wesentlich höherer Geschwindigkeit vonstatten als über die Standardwege von DOS (*Input* und *Output*).

Beispiel

```
uses Crt;
var
  f : Text;

begin
  Write('Ausgabe auf Bildschirm oder Drucker [B/D]? ');
  if UpCase(ReadKey) = 'D' then
     Assign(f, 'PRN')  { Ausgaben zum Drucker }
   else
     AssignCrt(f);  { sonst schnelle Bildschirmausgabe }

  Rewrite(f);
  Writeln(f, 'Eine Ausgabezeile...');
  Close(f);
end.
```

Bar-Prozedur — Graph

Funktion — zeichnet einen Balken.

Deklaration

```
Bar(x1, y1, x2, y2: Integer)
```

Erläuterung — *Bar* zeichnet ein gefülltes Rechteck, dessen obere linke Ecke durch die Koordinaten (*x1,y1*) und dessen untere rechte Ecke durch die Koordinaten (*x2,y2*) festgelegt ist.

Verwendet wird die momentan gesetzte Zeichenfarbe sowie das durch *SetFillStyle* bzw. *SetFillPattern* definierte Füll-Muster.

Für Fehler, die während des Ausfüllens auftreten, liefert *GraphResult* den Wert -6.

Grenzen — vor dem Aufruf von *Bar* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *Bar3D, GraphResult, SetFillStyle, SetFillPattern, SetLineStyle.*

Beispiel

```
program BarDemo;
uses Graph;
var
  GraphMode, GraphDriver: Integer;
  I, Breite: Integer;

begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
```

```
  Breite := 20;
  for I:= 1 to 9 do  { eine absteigende Treppe }
     Bar(I*Breite, I * 10; Succ(i)*Breite+1, 200);

  Readln;
  CloseGraph;
end.
```

Bar3D-Prozedur — Graph

Funktion — zeichnet einen dreidimensionalen Balken.

Deklaration

```
Bar3D(x1,y1,x2,y2: Integer; Depth: Word; Top: Boolean)
```

Erläuterung — Balken, die mit *Bar3D* gezeichnet werden, sind »dreidimensional« und bestehen aus einem gefüllten Rechteck, dessen obere linke Ecke durch den Punkt (*x1,y1*) und dessen untere rechte Ecke durch (*x2,y2*) festgelegt ist. *Bar3D* zeichnet die Umrißlinien in der durch *SetColor* gesetzten Farbe und der durch *SetLineStyle* festgelegten Linienart. Die umschlossene Fläche wird in der momentan gesetzten Zeichenfarbe und mit dem durch *SetFillStyle* bzw. *SetFillPattern* bestimmten Muster ausgefüllt, *GraphResult* liefert im Falle eines Fehlers den Wert -6.

Über den Parameter *Depth* läßt sich die räumliche Tiefe (in Pixeln) bestimmen, die typischerweise bei rund 25% der Breite liegt:

```
Bar3D(x1,y1,x2,y2, (x2-x1+1) div 4, TopOn);
```

Der Parameter *Top* legt fest, ob *Bar3D* einen oberen Abschluß des Balkens zeichnet oder nicht (im letzteren Fall lassen sich mehrere Balken übereinander stapeln). Für ihn sind zwei Konstanten in *Graph* definiert:

```
const
  TopOn = True;  { "Balkendeckel" wird gezeichnet }
  TopOff = False;  { nicht gezeichnet }
```

Grenzen — vor der Anwendung von *Bar* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *Bar, GraphResult, SetFillStyle, SetFillPattern, SetLineStyle.*

Beispiel

```
program Bar3Demo;
uses Graph;
var
  GraphMode, GraphDriver: Integer;
  y0,y1,y2,x1,x2: Integer;

begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
```

```
  y0 := 10; y1 := 60; y2 := 110;
  x1 := 10; x2 := 50;
  Bar3D(x1,y0,x2,y1, 10, TopOff);
  Bar3D(x1,y1,x2,y2, 10, TopOn);

  Readln;
  CloseGraph;
end.
```

BlockRead-Prozedur

Funktion — liest ein oder mehrere Records einer Datei in eine Puffervariable.

Deklaration

```
BlockRead(var f: File; var buf;
          count: Word [; result: Word])
```

Erläuterung — *f* steht für eine untypisierte Datei-Variable, *buf* für eine Variable beliebigen Typs (normalerweise ein *Byte*- oder *Char*-Array); *count* ist ein Ausdruck des Typs *Word*, *result* hat ebenfalls den Typ *Word* und muß eine Variable sein.

BlockRead liest *count* Records der durch *f* angegebenen Datei und speichert sie ab der Startadresse von *buf*. Gelesen wird ab der momentanen Position innerhalb von *f*. Wenn *result* als Parameter übergeben wurde, dann beendet *BlockRead* den Lesevorgang bei Erreichen des Dateiendes ohne Fehlermeldung und setzt *result* auf die Anzahl der gelesenen Records (*result* erhält also maximal den Wert von *count*); ist *result* dagegen nicht übergeben worden, dann erzeugt *BlockRead* bei Überschreitung des Dateiendes einen Laufzeitfehler.

Hinweis: result gibt die Anzahl der *komplett* gelesenen Records wieder. Wenn von einer Datei mit 128 Bytes Recordgröße 255 Bytes vor dem Erreichen des Dateiendes gelesen werden konnten, dann hat *result* also den Wert 1.

Die Gesamtzahl der zu übertragenden Bytes ergibt sich aus *count* * *RecSize*, wobei *RecSize* die bei der Eröffnung der Datei definierte Recordgröße ist (und von Turbo Pascal automatisch auf 128 gesetzt wird, wenn keine explizite Angabe erfolgt). Maximal können mit einem Aufruf von *BlockRead* 65535 Bytes (64 KByte) gelesen werden.

Die momentane Position innerhalb der Datei wird von *BlockRead* um *result* Records in Richtung auf das Dateiende versetzt.

Wenn *BlockRead* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* (im Modus `{$I-}`) den Wert 0; ansonsten hat diese Funktion einen Fehlercode <> 0 als Ergebnis.

Grenzen — die durch *f* angegebene Datei muß offen sein.

Querverweis *BlockWrite*

Beispiel

```
program Copy;
{ ein einfaches Kopierprogramm OHNE jede Fehlerprüfung }
var
  Quelle, Ziel: file;
  R_gelesen, R_geschrieben: Word;
  Puffer: array[1..2048] of Byte;

begin
  { Zuordnung und Öffnen der Quelldatei }
  Assign(Quelle, ParamStr(1));
  Reset(Quelle, 1);  { Recordlänge: 1 Byte }
  { Zuordnung und Erzeugen der Zieldatei }
  Assign(Ziel, ParamStr(2)); Rewrite(Ziel, 1);
  Writeln(FileSize(Quelle), ' Bytes werden kopiert...');
  repeat
   BlockRead(Quelle, Puffer, SizeOf(Puffer), R_gelesen);
   BlockWrite(Ziel, Puffer, R_gelesen, R_geschrieben);
  until (R_gelesen = 0) or (R_gelesen <> R_geschrieben);
  Close(Quelle); Close(Ziel);
end.
```

BlockWrite-Prozedur

Funktion schreibt ein oder mehrere Records aus einer Puffervariablen in eine Datei.

Deklaration

```
BlockWrite(var f: File; var buf;
           count: Word [; result: Word])
```

Erläuterung *f* steht für eine untypisierte Datei-Variable, *buf* für eine Variable beliebigen Typs (normalerweise ein *Byte*- oder *Char*-Array), *count* ist ein Ausdruck des Typs *Word*, *result* hat ebenfalls den Typ *Word* und muß eine Variable sein.

BlockWrite liest *count* Records ab der Startadresse von *buf* aus dem Speicher und schreibt sie in die durch *f* angegebene Datei. Geschrieben wird ab der momentanen Position innerhalb von *f*.

Wenn *result* als Parameter übergeben wurde, dann beendet *BlockWrite* den Lesevorgang auch dann ohne Fehlermeldung und setzt *result* auf die Anzahl der geschriebenen Records (*result* erhält also maximal den Wert von *count*), wenn aus irgendeinem Grund (am wahrscheinlichsten: Diskette voll) nicht sämtliche Records geschrieben werden konnten; ist *result* dagegen nicht übergeben worden, dann erzeugt *BlockWrite* in solchen Fällen einen Laufzeitfehler.

Hinweis: result gibt die Anzahl der *komplett* geschriebenen Records wieder. Wenn in eine Datei mit 128 Bytes Recordgröße

255 Bytes geschrieben werden konnten, dann hat *result* also den Wert 1.

Die Gesamtzahl der zu übertragenden Bytes ergibt sich aus *count* * *RecSize*, wobei *RecSize* die bei der Eröffnung der Datei definierte Recordgröße ist (und von Turbo Pascal automatisch auf 128 gesetzt wird, wenn keine explizite Angabe erfolgt). Maximal können mit einem Aufruf von *BlockWrite* 65535 Bytes (64 KByte) geschrieben werden.

Die momentane Position innerhalb der Datei wird von *BlockWrite* um *result* Records in Richtung auf das Dateiende versetzt.

Wenn *BlockWrite* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* (im Modus {$I-}) den Wert 0; ansonsten hat diese Funktion einen Fehlercode <>0 als Ergebnis.

Grenzen die durch *f* angegebene Datei muß offen sein.

Querverweis *BlockRead*

Beispiel siehe *BlockRead*

ChDir-Prozedur

Funktion wechselt das Standardverzeichnis.

Deklaration `ChDir(s: string)`

Erläuterung *s* ist ein Stringausdruck und gibt das neu zu setzende Standardverzeichnis an. Wenn *s* einen Laufwerksbezeichner enthält, wird das Laufwerk ebenfalls gewechselt. Wenn der durch *s* angegebene Suchweg mit einem umgekehrten Schrägstrich (backslash) beginnt, bezieht er sich auf das Stammverzeichnis des Laufwerks, ansonsten erfolgt der Wechsel relativ zum momentan gesetzten Verzeichnis.

Wenn *ChDir* fehlerfrei ausgeführt werden konnte, liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten liefert diese Funktion den entsprechenden Fehlercode (<>0) zurück.

Querverweis *GetDir, MkDir, RmDir*

Beispiel

```
begin {Verzeichnisname wird als erster Kommandozeilen-
       Parameter angenommen: }
  {$I-}
  ChDir(ParamStr(1));
  if IOResult <> 0 then
    Writeln('Directory kann nicht gesetzt werden!');
end.
```

Chr-Funktion

Funktion liefert ein Zeichen der angegebenen Ordinalität zurück.

Deklaration
```
Chr(x: Byte)
```

Ergebnistyp *Char*

Erläuterung *x* ist ein Integer-Ausdruck. Das Ergebnis der Funktion ist ein Zeichen, dessen ASCII-Code dem Wert von *x* entspricht.

Querverweis *Ord*

Beispiel
```
uses
  Printer
begin
  Write(lst, Chr(12)); { Ausgabe eines Seitenvorschubs }
end.
```

Circle-Prozedur Graph

Funktion zeichnet einen Kreis.

Deklaration
```
Circle(X,Y: Integer; Radius: Word)
```

Erläuterung *Circle* zeichnet einen Kreis mit der momentan gesetzten Farbe und dem durch *Radius* angegebenen Radius um den Mittelpunkt (*x*,*y*) herum.

Hinweis: Jeder Grafiktreiber definiert einen Wert (*AspectRatio*), der das tatsächliche Höhen-/Seitenverhältnis des Bildschirms wiedergibt und von *Circle*, *Arc* und *PieSlice* automatisch verwendet wird.

Grenzen vor dem Aufruf von *Circle* muß ein Grafikmodus gesetzt sein.

Querverweis *Arc*, *Ellipse*, *FillEllipse*, *GetArcCoords*, *GetAspectRatio*, *PieSlice*, *Sector*, *SetAspectRatio*.

Beispiel
```
program CircleDemo;
uses Graph;
var
  r, GraphMode, GraphDriver: Integer;

begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');

  for r := 1 to 5 do
    Circle(100,100, r * 10);  { fünf Kreise }

  Readln;
  CloseGraph;
end.
```

ClearDevice-Prozedur Graph

Funktion löscht den gesamten Grafikbildschirm.

Deklaration `ClearDevice`

Erläuterung *ClearDevice* löscht den gesamten Bildschirm unabhängig von einem eventuell gesetzten Zeichenfenster und setzt den Grafik-Cursor auf den Punkt (0,0) innerhalb des Zeichenfensters. Weitere Parameter (wie die Grenzen des Zeichenfensters etc.) werden nicht verändert.

Grenzen vor dem Aufruf muß ein Grafikmodus gesetzt worden sein.

Querverweis *ClearViewPort, CloseGraph, GraphDefaults, InitGraph, RestoreCrtMode, SetGraphMode*.

Beispiel

```
program ClearDeviceDemo;
uses Crt, Graph;
var
  GraphMode, GraphDriver: Integer;

procedure Lines;
var ch: Char;
    X, Y: Integer;
begin
    X := GetMaxX + 1; Y := GetMaxY + 1;
  repeat
    LineTo(Random(X), Random(Y));
  until Keypressed;
  ch := ReadKey;
end;

begin { Hauptprogramm }
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
  Lines;
  ClearDevice;  { löscht den gesamten Bildschirm }
  Lines;
  Readln;
  CloseGraph;
end.
```

ClearViewPort-Prozedur Graph

Funktion löscht den Inhalt des momentan gesetzten Zeichenfensters.

Deklaration `ClearViewPort`

Erläuterung *ClearViewPort* füllt das momentan gesetzte Zeichenfenster mit der durch *Palette(0)* gesetzten Farbe und setzt den Cursor auf die Koordinaten (0,0) innerhalb des Fensters.

Grenzen vor dem Aufruf von *ClearViewPort* muß ein Grafikmodus gesetzt worden sein.

Querverweis *SetViewPort, GetViewSettings.*

Beispiel
```
program ClearViewPortDemo;
uses Crt, Graph;
var
  GraphMode, GraphDriver: Integer;

begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
  Line(0,0,200,200);
  SetViewPort(10,10,150,100,ClipOn);
  OutText('<RETURN> drücken');
  Readln;
  ClearViewPort;
  OutTextXY(0,0,'Nochmal <RETURN>');
  Readln;
  CloseGraph;
end.
```

Close-Prozedur

Funktion schließt eine Datei.

Deklaration `Close(var f)`

Erläuterung *f* ist eine Datei-Variable beliebigen Typs und mit einer externen Datei verbunden, die zuvor mit *Reset*, *Rewrite* oder *Append* eröffnet wurde. Diese Datei wird auf den neuesten Stand gebracht, über einen Aufruf von *Flush* wird ein Schreiben eventuell noch »ungeschriebener« Daten erzwungen; danach gibt *Close* den DOS-Handle wieder frei.

Hinweis: Die Verbindung zwischen der Datei-Variablen und dem mit *Assign* zugeordneten Namen wird durch *Close* nicht aufgehoben.

Wenn *Close* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus `{$I-}` den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Querverweis *Append, Assign, Reset, Rewrite*

Beispiel
```
program CloseDemo;
var f: file;
begin
  Assign(f, '\AUTOEXEC.BAT');
  Reset(f, 1);                { Recordgröße 1 }
```

```
  Writeln('AUTOEXEC.BAT ist ',
          FileSize(f), ' Bytes lang.');
  Close(f);
end.
```

CloseGraph-Prozedur — Graph

Funktion — beendet das Grafikpaket.

Deklaration — `CloseGraph`

Erläuterung — *CloseGraph* beendet das Grafikpaket und schaltet in den Textmodus zurück, der vor der Umschaltung zur Grafik aktiv war. Dynamisch belegte Puffer sowie auf den Heap geladene Grafiktreiber und Zeichensätze werden über Aufrufe von *GraphFreeMem* wieder freigegeben.

Grenzen — vor dem Aufruf von *CloseGraph* muß ein Grafik-Modus gesetzt worden sein.

Querverweis — *DetectGraph*, *GetGraphMode*, *InitGraph*, *RestoreCrtMode*, *SetGraphMode*.

Beispiel

```
program  CloseGraphDemo;
var
  GraphDriver, GraphMode: Integer;
begin
  GraphDriver := Detect;   { automatische Erkennung }
  InitGraph(GraphDriver, GraphMode, '\TP');  { Start }

  { ...    Grafik-Operationen }
  Readln;
  CloseGraph;   { beendet das Grafikpaket }
end.
```

ClrEol-Prozedur — Crt

Funktion — löscht alle Zeichen von der momentanen Position des Cursors bis zum Zeilenende.

Deklaration — `ClrEol`

Erläuterung — *ClrEol* füllt die Spalten rechts der momentanen Cursorposition bis zum Ende der momentanen Zeile mit Leerzeichen und verwendet dazu das von *TextBackground* gesetzte Attribut – wenn hier nicht »schwarz« gesetzt ist, wird die Zeile entsprechend eingefärbt. Die Position des Cursors selbst bleibt unverändert.

ClrEol arbeitet relativ zu einem eventuell gesetzten Textfenster. Eine Befehlsfolge wie

```
Window(1,5,60,20);
GotoXY(3,5);
ClrEol;
```

löscht also die Spalten 4 bis 60 der zehnten Bildschirmzeile.

Querverweis *ClrScr, Window.*

Beispiel
```
uses Crt;
begin
  TextBackground(LightGray);
  ClrEol; { färbt die oberste Bildschirmzeile hellgrau }
end.
```

ClrScr-Prozedur — Crt

Funktion löscht den Bildschirm und setzt den Cursor in die obere linke Ecke.

Deklaration `ClrScr`

Erläuterung *ClrScr* füllt den Bildschirm mit Leerzeichen und benutzt dazu das mit *TextBackground* gesetzte Attribut – wenn hier nicht »schwarz« gesetzt ist, wird der Bildschirm entsprechend eingefärbt. (Das gilt übrigens auch für Bildschirmteile, die mit *ClrEol, InsLine* und *DelLine* verändert werden).

ClrScr arbeitet relativ zu einem eventuell gesetzten Textfenster. Eine Befehlsfolge wie

```
Window(20,7,60,18);
TextBackground(LightGray);
ClrScr;
```

erzeugt also ein hellgraues Rechteck in der Mitte des Bildschirms und setzt den Cursor in die 20. Spalte der 7. Zeile. Bildschirmteile außerhalb des momentan gesetzten Fensters bleiben von *ClrScr* unbeeinflußt.

Querverweis *ClrEol, Window.*

Beispiel
```
uses Crt;
begin
  TextBackground(Red);
  ClrScr;   { färbt den gesamten Bildschirm rot }
end.
```

Concat-Funktion

Funktion verbindet zwei oder mehrere Stringteile miteinander.

Deklaration `Concat(s1 [, s2, ..., sn]: string)`

Ergebnistyp *String*

Erläuterung Jeder Parameter von *Concat* ist ein String-Ausdruck. Das Ergebnis der Funktion ist ein String, der sich aus allen übergebenen Argumenten in der Reihenfolge ihrer Angabe zusammensetzt und automatisch nach dem 255. Zeichen abgeschnitten wird.

Hinweis: In Turbo Pascal hat der Operator + dieselbe Wirkung wie die Standardfunktion *Concat*.

Querverweis *Copy, Delete, Insert, Length, Pos*

Beispiel
```
var
  s: string;
begin
  s := Concat('ABC', 'DEF');  { s := 'ABCDEF'}
  s := 'ABC' + 'DEF'    { hat dasselbe Ergebnis }
end.
```

Copy-Funktion

Funktion liefert einen Teil eines Strings zurück.

Deklaration `Copy(s: string; index, count: Integer)`

Ergebnistyp *String*

Erläuterung *Copy* liefert *count* Zeichen ab der durch *index* angegebenen Position des Strings *s* zurück. *s* ist ein String-Ausdruck, *count* und *index* sind Integer-Ausdrücke.

Wenn *index* größer als die momentane Länge von *s* ist, liefert *Copy* einen Nullstring; wenn *s* kürzer als *index* + *count* ist, werden nur die Zeichen zurückgeliefert, die *s* ab der durch *index* angegebenen Position enthält.

Querverweis *Concat, Insert, Delete, Length, Pos*

Beispiel
```
var
  s : string;
begin
  s := 'ABCDEF';
  Writeln(Copy(s,2,3));  { 'BCD');
  Writeln(Copy(s,100,10));  { '' }
end.
```

Cos-Funktion

Funktion liefert den Cosinus des Arguments zurück.

Deklaration `Cos(x: Real)`

Ergebnistyp *Real*

Erläuterung *x* ist ein Real-Ausdruck und wird als Winkel in der Einheit *Rad* (0..2 Pi =0..360 Grad) interpretiert. Das Ergebnis der Funktion ist der Cosinus von *x*.

Querverweis *ArcTan*, *Sin*

Beispiel
```
var r: Real;
begin
  r := Cos(Pi);   { Ergebnis -1 }
end.
```

CSeg-Funktion

Funktion liefert die Adresse des momentanen Code-Segments zurück.

Deklaration `CSeg`

Ergebnistyp *Word*

Erläuterung *CSeg* liefert den momentanen Inhalt des Prozessor-Registers CS zurück – also die Adresse des Segments, in dem der Code des »momentanen« Moduls gespeichert ist.

Querverweis *DSeg*, *SSeg*

Dec-Prozedur

Funktion erniedrigt eine Variable.

Deklaration `Dec(x [, n: LongInt])`

Erläuterung *x* ist eine Variable ordinalen Typs, der optionale Parameter *n* ist ein Integer-Ausdruck. Wenn *n* nicht angegeben ist, erniedrigt *Dec* die Variable *x* um 1 (und entspricht damit der Zuweisung `x := x - 1`), ansonsten wird *x* um *n* erniedrigt (`x := x - n`).

Dec ist eine Art **inline**-Prozedur und erzeugt optimierten Code. Die Vorteile dieser »Abkürzung« kommen am meisten in engen Schleifen und bei komplexen Adressierungen zum Tragen.

Querverweis *Inc*, *Pred*

Beispiel
```
var
  IntVar : Integer;
  LongVar: LongInt;
begin
  IntVar := 10; LongVar := 90234;
  Dec(IntVar);     { IntVar := IntVar - 1    (= 9) }
  Dec(LongVar,5);  { LongVar := LongVar - 5  (= 90229) }
end.
```

Delay-Prozedur — Crt

Funktion — wartet die angegebene Anzahl von Millisekunden.

Deklaration
```
Delay(ms: Word)
```

Erläuterung — *ms* ist ein Ausdruck des Typs *Word* und gibt die Anzahl von Millisekunden an, um die verzögert werden soll.

Delay arbeitet im Gegensatz zu einer normalen Leerschleife unabhängig vom verwendeten Computermodell. Beim Start des Programms wird mit Hilfe der eingebauten Uhr festgestellt, wie schnell der Prozessor arbeitet und ein entsprechender Zähler für eine Verzögerungsschleife festgelegt.

Dieser Zähler ist eine Annäherung: beispielsweise läßt ein Aufruf wie *Delay(60000);* den Computer je nach Modell zwischen 59 und 61 Sekunden warten – hundertprozentig genau ist *Delay* also nicht.

Querverweis — *Sound*, *NoSound*.

Delete-Prozedur

Funktion — löscht einen Teil eines Strings.

Deklaration
```
Delete(var s: string; index , count: Integer)
```

Erläuterung — *s* ist eine String-Variable, *index* und *count* sind Integer-Ausdrücke. *Delete* löscht ab der durch *index* angegebenen Position *count* Zeichen aus dem String *s* heraus. Wenn *index* größer als die momentane Länge von *s* ist, passiert gar nichts; wenn *s* kürzer oder genauso lang wie *index* + *count* ist, läßt *Delete* nur die vor der Position *index* stehenden Zeichen übrig.

Querverweis — *Concat*, *Copy*, *Insert*, *Length*, *Pos*

Beispiel
```
var  s: string
begin
  s := 'ABCDEF';
  Delete (s,3,1);  { 'ABDEF' }
  Delete (s,4,99);  { 'ABD' }
end.
```

DelLine-Prozedur — Crt

Funktion — entfernt eine Zeile vom Bildschirm.

Deklaration
```
DelLine
```

Erläuterung — *DelLine* löscht die Zeile, auf der sich der Cursor befindet, und rollt danach (über die BIOS-Routinen des Computers) alle folgenden Zeilen um eine Position nach oben. Die dadurch

freiwerdende unterste Zeile wird gelöscht (d.h. mit Leerzeichen gefüllt, die das mit *TextBackground* gesetzte Attribut haben).

DelLine arbeitet relativ zu einem eventuell gesetzten Textfenster. Eine Befehlsfolge wie

```
Window(1,10,60,20);
GotoXY(1,1);
DelLine;
```

löscht die oberste Zeile des Fensters (also die Spalten 1 bis 60 der zehnten Bildschirmzeile).

Außerhalb des Fensters liegende Teile des Bildschirms bleiben nicht nur von der Löschung der Zeile, sondern auch vom Rollen des Fensterinhalts unberührt.

Querverweis *InsLine, Window.*

Beispiel

```
uses Crt;
var
  x: Integer;
begin
  ClrScr;  { löscht den Bildschirm }
  for x:= 1 to 3 do
    Writeln(x);
  Readln;
  GotoXY(50,2);  { Cursor irgendwo in die Zeile 2 }
  DelLine;
end.
```

DetectGraph-Prozedur — Graph

Funktion prüft die Hardware des Computers und bestimmt, welcher Grafiktreiber und welcher Grafikmodus verwendet werden sollen.

Deklaration `DetectGraph(var GraphDriver , GraphMode: Integer)`

Erläuterung *DetectGraph* wird normalerweise von *InitGraph* aufgerufen und stellt fest, mit welchem Adapter der Computer ausgerüstet ist – *InitGraph* lädt dann den entsprechenden Treiber. Wenn *DetectGraph* keinen grafikfähigen Adapter erkennen kann, liefert diese Routine genauso wie ein folgender Aufruf von *GraphResult* das Ergebnis -2.

Die folgenden Ergebnis-Konstanten für *DetectGraph* sind im Unit *Graph* definiert:

```
const
  Detect   = 0;    { automatische Erkennung }
  CGA      = 1;
  MCGA     = 2;
```

```
EGA      = 3;
EGA64    = 4;
EGAMono  = 5;
IBM8514  = 6;
HercMono = 7;
ATT400   = 8;
VGA      = 9;
PC3270   = 10;
```

Es gibt überhaupt nur einen Grund, *DetectGraph* direkt aufzurufen bzw. die Ergebnisse dieser Routine zu simulieren – nämlich dann, wenn Sie absichtlich einen bestimmten Modus setzen wollen. Das Beispielprogramm in diesem Abschnitt prüft, ob der Computer mit einem Color Graphics Adapter (CGA) oder einer EGA-Karte (64 KByte oder 256 KByte) ausgerüstet ist – in allen drei Fällen wird danach der CGA-Grafikmodus gesetzt.

Achtung: Wenn *InitGraph* für *GraphDriver* ein anderer Wert als *Detect* übergeben wird, muß auch der zweite Parameter (*GraphMode*) definiert sein.

Grenzen

der IBM 8514-Adapter emuliert im »Normalzustand« eine VGA-Karte – und genauso wird er auch von *DetectGraph* behandelt.

Wenn die speziellen Eigenschaften dieses Adapters genutzt werden sollen, muß der Aufruf von *InitGraph* deshalb mit den entsprechenden Werten für *GraphMode* und *GraphDriver* erfolgen.

Querverweis

InitGraph, InstallUserDriver, GraphResult.

Beispiel

```
uses Graph;
var
  GraphDriver, GraphMode: Integer;

begin
  DetectGraph(GraphDriver, GraphMode);
  if (GraphDriver = EGA) or
     (GraphDriver = EGA64) or
     (GraphDriver = CGA) then
     begin
       GraphDriver := CGA;
       GraphMode := CGAHi;
       InitGraph(GraphDriver, GraphMode, '\TP');
     end
    else begin
       Writeln ('Fehler! Programm benötigt ...');
   ...
```

DiskFree-Funktion — Dos

Funktion — liefert die Größe des freien Speicherplatzes auf einem Laufwerk zurück.

Deklaration
```
DiskFree(Drive: Byte)
```

Ergebnistyp — *LongInt*

Erläuterung — *Drive* ist ein Ausdruck des Typs *Byte* und gibt das Laufwerk an, dessen freier Speicherplatz (in Bytes) ermittelt werden soll: der Wert 0 steht für das momentan gesetzte Laufwerk, der Wert 1 für Laufwerk A:, 2 für B: usw. *DiskFree* liefert -1 als Funktionsergebnis zurück, wenn *Drive* einen ungültigen Wert hat.

Querverweis — *DiskSize*, *GetDir*

Beispiel
```
uses Dos;
begin
  Writeln('Auf dem momentan gesetzten Laufwerk sind ',
          DiskFree(0) div 1024, 'KByte verfügbar.');
end.
```

DiskSize-Funktion — Dos

Funktion — liefert die Gesamtkapazität eines Laufwerks zurück.

Deklaration
```
DiskSize(Drive: Byte)
```

Ergebnistyp — *LongInt*

Erläuterung — *Drive* ist ein Ausdruck des Typs *Byte* und gibt das Laufwerk an, dessen Kapazität (in Bytes) ermittelt werden soll. Der Wert 0 steht für das momentan gesetzte Laufwerk, der Wert 1 für Laufwerk A:, 2 für B: usw. *DiskSize* liefert -1 als Funktionsergebnis zurück, wenn *Drive* einen ungültigen Wert hat.

Querverweis — *DiskFree*, *GetDir*

Beispiel
```
uses Dos;
begin
  Writeln('Das momentan gesetzte Laufwerk hat eine ',
     'Kapazität von ', DiskSize(0) div 1024, 'KByte.');
end.
```

Dispose-Prozedur

Funktion — gibt den durch eine dynamische Variable belegten Speicherplatz auf dem Heap wieder frei.

Deklaration
```
Dispose(var p:Pointer)
Dispose(var p:Pointer, Destrukt)
```

Erläuterung In der ersten Form der Deklaration ist *p* eine Zeigervariable beliebigen Typs, der zuvor mit *New*, *GetMem* oder durch eine direkte Zuweisung ein Speicherbereich auf dem Heap zugeordnet wurde.

Speziell für objektorientierte Programmierung wurde die Syntax von *Dispose* erweitert. In der zweiten Form der Deklaration ist *p* eine Zeigervariable auf einen Objekttyp und *Destrukt* ein Aufruf der Destruktor-Methode dieses Objekttyps. Diese erweiterte Syntax von *Dispose* ist nichts anderes als eine Zusammenfassung der beiden Anweisungen:

```
P^.Destrukt;
Dispose(P);
```

Die erweiterte Syntax verbessert die Lesbarkeit und erzeugt außerdem einen kürzeren und effektiveren Code.

Achtung: Der Wert von *p* ist nach dem Aufruf von *Dispose* nicht mehr definiert, Zugriffe der Form *p^* stellen einen Fehler dar.

Grenzen wenn *p* nicht auf eine Adresse innerhalb des Heap-Bereichs zeigt, bricht das Programm mit einem Laufzeitfehler ab.

Dispose und *FreeMem* können nur unter bestimmten Umständen zusammen mit *Mark* und *Release* verwendet werden (siehe Kapitel 14 »Interne Details« des *Programmierhandbuches*).

Querverweis *New*, *GetMem*, *FreeMem*, *Release*

Beispiel

```
type Str22 = string[22];
var  p : ^Str22;
begin
  New(p);
  p^:= 'Hier ist der Text...';
  Writeln(p^);
  Dispose(p);   { 'Weg ist der Text ... ' }
end.
```

DosExitCode-Funktion — Dos

Funktion liefert den Exit-Code eines als Unterprozeß gestarteten Programms zurück.

Deklaration `DosExitCode`

Ergebnistyp *Word*

Erläuterung das niederwertige Byte des Ergebnisses ist der Exit-Code des Programms (0 für normales Ende, beliebige Werte für ein Ende via *Halt* usw.); das höherwertige Byte kann die folgenden Werte annehmen:

0 normales Ende
1 Abbruch durch Ctrl C oder Ctrl Break
2 Abbruch durch einen Gerätefehler (device error)
3 Ende durch die Prozedur Keep

Querverweis *Exec*

DosVersion-Funktion — Dos

Funktion liefert die Versionsnummer von DOS zurück.

Deklaration `DosVersion`

Ergebnistyp *Word*

Erläuterung *DosVersion* liefert die Versionsnummer von DOS in Form eines *Word*-Wertes zurück, dessen Bytes voneinander getrennt interpretiert werden müssen: das höherwertige Byte enthält die Neben-Versionsnummer, das niederwertige Byte die Haupt-Versionsnummer. Für die DOS-Version 3.20 würde *DosVersion* also beispielsweise das Ergebnis $2003 zurückliefern.

Beispiel

```
program DosVersionDemo;
uses Dos;
var
  Version: Word;

begin
  Version := DosVersion;
  Writeln('Dieses Programm läuft unter der DOS-',
          'Version ', Lo(Version), '.', Hi(Version));
end.
```

DrawPoly-Prozedur — Graph

Funktion zeichnet den Umriß eines Polygons.

Deklaration `DrawPoly(NumPoints: Word; var PolyPoints)`

Erläuterung *PolyPoints* ist ein untypisierter Parameter, der die Koordinaten der Eckpunkte des Polygons enthält. Die Anzahl der Koordinatenpaare wird in *NumPoints* angegeben (einem Ausdruck des Typs *Word*). Für Koordinatenpaare definiert das Unit *Graph* den Datentyp *PointType* als:

```
type
  PointType = record
                x, y: Word;
              end;
```

DrawPoly verwendet die momentan gesetzte Zeichenfarbe und die durch *SetLineStyle* gesetzte Linienart, wobei der jeweils

letzte Aufruf von *SetWriteMode* festlegt, ob die Linien direkt oder über eine XOR-Operation mit dem vorherigen Bildschirm gezeichnet werden.

Hinweis: DrawPoly beginnt mit dem ersten angegebenen Punkt und zieht von dort aus eine Linie zum zweiten, von da zum dritten usw. Zum Zeichnen einer geschlossenen Figur mit *n* Eckpunkten muß *PolyPoints n*+1 Koordinatenpaare enthalten und das letzte Paar (*PolyPoints*(*n*+1)) wieder mit dem ersten identisch sein (siehe »Beispiel«).

Grenzen vor dem Aufruf von *DrawPoly* muß ein Grafikmodus gesetzt worden sein.

Querverweis *FillPoly, GetLineSettings, SetColor, SetLineStyle, GraphResult.*

Beispiel

```
program DrawPolyDemo;
uses Crt, Graph;
const
  Dreieck: array[1..4] of PointType =
   ( (x: 50;  y: 100),
     (x: 100; y: 100),
     (x: 150; y: 150),
     (x: 50;  y: 100));
var
  GraphMode, GraphDriver: Integer;

begin
  GraphDriver := Detect;  { automatische Erkennung }
  InitGraph(GraphDriver, GraphMode, '\TP');

  DrawPoly(SizeOf(Dreieck) div SizeOf(PointType),
           Dreieck);
  Readln;
  CloseGraph;
end.
```

DSeg-Funktion

Funktion liefert die Segmentadresse des Daten-Segments zurück.

Deklaration `DSeg`

Ergebnistyp *Word*

Erläuterung *DSeg* liefert den Inhalt des Prozessor-Registers DS zurück, d.h. die Adresse des Segments, in dem die globalen Variablen des Programms gespeichert sind.

Hinweis: Die Anwendung von *Seg* auf eine globale Variable hat dasselbe Ergebnis.

Querverweis *CSeg, SSeg*

Beispiel

```
begin
  Writeln('Die globalen Variablen dieses Programms ',
          'stehen ab der Adresse ',
          DSeg, ':0000 im Speicher.');
end.
```

Ellipse-Prozedur — Graph

Funktion — zeichnet einen elliptischen Kreisausschnitt.

Deklaration

```
Ellipse(X,Y: Integer; StAngle , EndAngle: Word;
        XRadius, YRadius: Word;)
```

Erläuterung — *Ellipse* zeichnet einen elliptischen Kreisbogen um den Mittelpunkt (*X,Y*) mit dem Radius *XRadius* (horizontal) und *YRadius* (vertikal). Alle Parameter dieser Funktion können sowohl Variablen als auch Ausdrücke sein.

StAngle legt den Startpunkt, *EndAngle* den Endpunkt fest. Die Angabe beider Winkel erfolgt in Grad, wobei entgegen dem Uhrzeigersinn gezählt wird (0 Grad = horizontal rechts des Mittelpunkts, 90 Grad = senkrecht über dem Mittelpunkt usw). Eine Ellipse von *StAngle* = 0 bis *EndAngle* = 360 ergibt einen vollen Kreis.

Ellipse benutzt die momentan gesetzte Farbe und die durch *SetLineStyle* gesetzte Linienart; wie bei *Circle* und *Arc* wird das Höhen-/Seitenverhältnis des Bildschirms automatisch berücksichtigt.

Grenzen — vor dem Aufruf dieser Prozedur muß ein Grafikmodus gesetzt worden sein.

Querverweis — *Circle, Arc, PieSlice, GetArcCoords, GetAspectRatio, FillEllipse, Sector, SetAspectRatio.*

Beispiel

```
program EllipseDemo;
uses Crt, Graph;
var
  GraphMode, GraphDriver: Integer;
  XR, YR, i: Integer;
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
  YR := 110; XR:= 10;
  for i := 1 to 10 do begin
    SetColor(i); Ellipse(100,100,0,360,XR,YR);
    Inc(XR, 10); Dec(YR, 10);
   end;
  Readln;
  CloseGraph;
end.
```

EnvCount-Funktion — Dos

Funktion — liefert die Anzahl der Einträge in der Tabelle *Environment* zurück.

Deklaration
```
EnvCount
```

Ergebnistyp — *Integer*

Erläuterung — *EnvCount* analysiert die Tabelle *Environment*, die jedem Programm von DOS automatisch vorangestellt wird und eine unsortierte Folge von Einträgen der Form *Name=Text* enthält. Zurückgeliefert wird die Anzahl dieser Einträge, auf die dann mit der Funktion *EnvStr* zugegriffen werden kann.

Querverweis — *EnvStr*, *GetEnv*, *ParamCount*, *ParamStr*

Beispiel
```
program EnvCountDemo;
uses Dos;
var
  I: Integer;
begin
  for I:= 1 to EnvCount do
    Writeln(EnvStr(I));
end.
```

EnvStr-Funktion — Dos

Funktion — liefert einen Eintrag der DOS-Tabelle *Environment*.

Deklaration
```
EnvStr(Index: Integer)
```

Ergebnistyp — *String*

Erläuterung — *Index* ist ein Ausdruck des Typs *Integer* und gibt die Nummer des *Environment*-Eintrags an, der von *EnvStr* zurückgeliefert werden soll. Das Ergebnis von *EnvStr* ist der komplette Eintrag, d.h. ein String der Form *Name=Text*.

Wenn *Index* den Wert 0 hat oder größer als *EnvCount* ist, dann liefert *EnvStr* einen Nullstring zurück.

Querverweis — *EnvCount*, *GetEnv*, *ParamStr*

Eof-Funktion

Funktion — prüft, ob das Ende einer Datei erreicht ist.

Deklaration

`Eof(var f)`	für typisierte und untypisierte Dateien
`Eof[(var f: Text)]`	für Textdateien

Ergebnistyp — *Boolean*

Erläuterung

Eof prüft, ob es in der angegebenen Datei ohne explizite Veränderung der momentanen Position »noch etwas zu lesen gibt«. In den folgenden Fällen wird eine Datei als »ausgelesen« betrachtet und *Eof* liefert das Ergebnis *True*:

- die Datei enthält überhaupt keine Komponenten (d.h. hat einen Umfang von 0 Bytes);
- die Datei ist nur für Schreibaktionen eröffnet worden oder es handelt sich um ein DOS-Gerät, das keine Leseaktionen zuläßt (wie beispielsweise der Bildschirm);
- der Positionszeiger innerhalb der Datei steht hinter der letzten Komponente.

Wenn *f* nicht angegeben ist, prüft Turbo Pascal die Standard-Eingabe: der Aufruf von *Eof* ohne Argument ist also identisch mit *Eof*(*Input*). Wenn die Prüfung durch *Eof* fehlerfrei ausgeführt werden konnte, liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Querverweis

Eoln, *SeekEof*

Beispiel

```
program EofDemo;
{ Programm zur unformatierten Ausgabe von Textdateien }

var
  f : Text;
  ch : Char;

begin
  { Dateiname wird als 1. Kommandozeilen-
    Parameter vorausgesetzt }
  Assign(f, ParamStr(1));
  Reset(f);
  while not Eof(f) do  { solange das Dateiende
                         nicht erreicht ist }
  begin
   Read(f, ch); Write(ch);  { Ausgabe }
  end;
  Close(f);
end.
```

Eoln-Funktion

Funktion

prüft, ob die Position innerhalb einer Datei auf einem Zeilenende steht.

Deklaration

`Eoln(var f)`	für typisierte und untypisierte Dateien
`Eoln[(var f:Text)]`	für Textdateien

Ergebnistyp *Boolean*

Erläuterung *Eoln* liefert *True* zurück, wenn der Positionszeiger innerhalb der durch *f* angegebenen Datei auf eine Zeilenende-Markierung zeigt oder das Dateiende erreicht ist (*Eof* = *True*).

Wenn *f* nicht angegeben ist, prüft Turbo Pascal die Standard-Eingabe: der Aufruf von *Eoln* ohne Argument ist also identisch mit *Eoln*(*Input*).

Wenn die Prüfung durch *Eoln* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus `{$I-}` den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Querverweis *Eof*, *SeekEoln*

Erase-Prozedur

Funktion löscht eine Datei.

Deklaration `Erase(var f)`

Erläuterung *Erase* löscht die mit *f* (via *Assign*) verbundene externe Datei (wobei *f* für eine Datei-Variable beliebigen Typs steht).

Wenn *Erase* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus `{$I-}` den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Grenzen *Erase* darf nie auf offene Dateien angewendet werden.

Querverweis *Rename*

Beispiel

```
program EraseDemo;
  { Programm zum Löschen einer Datei OHNE Rückfrage }
var
  f : file;
begin
  { Name der zu löschenden Datei wird
    als 1.Kommandozeilen-Parameter vorausgesetzt }
  Assign(f, ParamStr(1));
  {$I-} Reset(f); {$I+}
  if IOResult <> 0 then
    Writeln('Nicht gefunden: ', ParamStr(1))
   else begin
    Close(f);   { Datei darf NICHT offen sein }
    Erase(f);
   end;
end.
```

Exec-Prozedur — Dos

Funktion — führt ein Programm als Subprozeß aus und übergibt Kommandozeilen-Parameter.

Deklaration

```
Exec(Path, CmdLine: string)
```

Erläuterung — *Path* gibt den Namen des Programms an und kann optional einen Laufwerksbezeichner und/oder einen Suchweg enthalten, durch *CmdLine* werden (optional) Kommandozeilen-Parameter übergeben. DOS-Befehle wie DIR, die über COMMAND.COM ausgeführt werden, müssen COMMMAND.COM als *Path* angeben, eventuelle Parameter werden mit */C* eingeleitet. Ein Beispiel:

```
Exec('\COMMAND.COM', '/C DIR *.PAS');
```

Hinweis: Eine automatische Bestimmung der Dateiart (.EXE, .COM, .BAT) findet nicht statt, das Suffix muß also explizit angegeben werden.

Fehler bei der Ausführung werden in der globalen Variablen *DosError* gespeichert, die möglichen Codes sind 2, 8, 10 und 11. Der Exit-Code des Programms läßt sich nach dem Aufruf von *Exec* über die Funktion *DosExitCode* ermitteln.

Achtung: Exec hat keinen Einfluß auf die dynamische Speicherverwaltung. Bei der Compilierung eines Programms, das diese Prozedur benutzt, muß also auf jeden Fall eine Maximalgröße des Heaps festgelegt werden – ansonsten ist der gesamte Speicher belegt (und DOS reagiert mit dem Fehlercode 8 – »kein Platz im Hauptspeicher«).

Achtung: Turbo Pascal-Programme besetzen – speziell bei Verwendung des Emulators – eine Reihe von Interrupt-Vektoren des Systems. Damit ein via *Exec* aufgerufenes Programm diese Vektoren nicht versehentlich als »Originale« benutzt oder gar verändert, sollte direkt vor und nach *Exec* ein Aufruf der Prozedur *SwapVectors* programmiert werden:

```
SwapVectors; { setzt die Originalvektoren des Systems }
Exec(....);  { Aufruf des neuen Programms }
SwapVectors; { setzt wieder die "Turbo-Vektoren" }
```

Grenzen — der von *Exec* ausgeführte DOS-Aufruf wird auf Novell-Netzwerken erst ab der Version 2.01-2 unterstützt.

Querverweis — *DosExitCode, SwapVectors*

Beispiel

```
program ExecDemo;
{ Demonstriert die Ausführung eines anderen Programms
  und kann übrigens auch verwendet werden, um sich
  sozusagen selbst aufzurufen }

{$M $4000, 0, 0}  { 16 KByte Stack, 0 Byte Heap }
```

```
uses Dos;
var
 Programmname, Kommandozeile: string;

begin
  Write ('Name des auszuführenden Programms: ');
  Readln(Programmname);
  Write ('Kommandozeilen-Parameter für ',
         Programmname, ': ');
  Readln(Kommandozeile);
  Writeln('Programm wird ausgeführt...');
  SwapVectors;   { 1. Austausch SaveIntXX <-> System }
  Exec(Programmname, Kommandozeile);
  SwapVectors;  { 2. Austausch SaveIntXX <-> System -
                    setzt wieder die "Turbo-Vektoren" }
  Writeln('Wir sind wieder im Programm ExecDemo!');
  if DosError <> 0 then    { Fehler ?}
    Writeln('Dos-Fehler: ', DosError)
   else
    Writeln('Fehlerfreie Ausführung. ',
             'Exit-Code: ', DosExitCode);
end.
```

Exit-Prozedur

Funktion verläßt den momentanen Block mit sofortiger Wirkung.

Deklaration `Exit`

Erläuterung die Ausführung von *Exit* innerhalb eines Unterprogramms (d.h. einer Prozedur oder Funktion) ist mit einem Rücksprung zur aufrufenden Routine identisch. Die Ausführung von *Exit* innerhalb des Hauptprogramms beendet das gesamte Programm. *Exit* hat im Prinzip denselben Effekt wie ein **goto** zu einem Label, das direkt vor dem jeweiligen Block-Ende steht.

Querverweis *Halt*

Beispiel

```
uses Crt;
procedure Warten;
begin
  repeat
   if KeyPressed then Exit;
   Write('Xx');
  until False;  { endlos }
end;

begin
  Warten;
end.
```

Exp-Funktion

Funktion liefert *e* hoch dem Argument zurück.

Deklaration `Exp(x: Real)`

Ergebnistyp *Real*

Erläuterung *x* ist ein Real-Ausdruck. Das Ergebnis der Funktion ist e^x, wobei *e* die Basis des natürlichen Logarithmus ist (2.718281...).

Querverweis *Ln*

Fail-Prozedur

Funktion gibt die dynamische Instanz eines Objekttyps wieder frei und verläßt den Konstruktor.

Deklaration `Fail`

Erläuterung Diese Prozedur kann nur innerhalb einer Konstruktor-Methode aufgerufen werden. *Fail* gibt die dynamische Instanz eines Objekttyps, die beim Einsprung in den Konstruktor angelegt wurde, wieder frei und verläßt anschließend sofort den Konstruktor.

Beispiel

```
type
  LinePtr = ^Line;
  Line = string[79];
  Base = object
    L1, L2: LinePtr;
    constructor Init(S1, S2 : Line);
    destructor Done; virtual;
    procedure Dump; virtual;
end;

constructor Base.Init(S1, S2 : Line);
begin
  New(L1);
  New(L2);
  if (L1 = nil) or (L2 = nil) then
  begin
    Base.Done;          { L1 und L2 freigeben }
    Fail;             { Base-Objekt freigeben }
  end;

  L1^ := S1;
  L2^ := S2;
end;
```

FExpand-Funktion — Dos

Funktion — erweitert einen unvollständig angegebenen Dateinamen um den dazugehörigen Suchweg.

Deklaration
```
FExpand(Path: PathStr)
```

Ergebnistyp — *PathStr*

Erläuterung — *FExpand* bildet aus einem unvollständigen Dateinamen und dem momentan gesetzten Verzeichnis einen vollständigen Namen bzw. Suchweg: der als *Path* übergebene Stringausdruck wird in Großbuchstaben »übersetzt«, davor kommt der Bezeichner des aktuellen Laufwerks, ein Doppelpunkt, ein umgekehrter Schrägstrich und ein Suchweg, der sich auf das Stammverzeichnis dieses Laufwerks bezieht. Die Angaben . und .. werden genauso wie Jokerzeichen korrekt behandelt.

Wenn beispielsweise momentan das Verzeichnis C:\TURBO\DOC gesetzt ist, dann liefert *FExpand* die folgenden Ergebnisse:

```
FExpand('test.pas')     =    'C:\TURBO\DOC\TEST.PAS'
FExpand('..\*.tpu')     =    'C:\TURBO\*.TPU'
FExpand('c:\bin\*.txt') =    'C:\BIN\*.TXT'
```

Der Datentyp *PathStr* ist in der Unit *Dos* als *string*[79] deklariert.

Hinweis: Vollständige Dateinamen/Suchwege lassen sich mit der Prozedur *FSplit* in ihre Komponenten »Suchweg«, »Dateiname« und »Suffix« zerlegen.

Querverweis — *FindFirst*, *FindNext*, *FSplit*

FilePos-Funktion

Funktion — liefert die momentane Position innerhalb einer Datei zurück.

Deklaration
```
FilePos(var f)
```

Ergebnistyp — *LongInt*

Erläuterung — *f* steht für eine typisierte oder untypisierte Datei-Variable. Wenn die Position am Anfang der Datei steht (d.h. vor der ersten Komponente), dann liefert *FilePos* den Wert 0 zurück; steht sie auf dem Ende der Datei (*Eof* = *True*), dann ist *FilePos*(*f*) identisch mit *FileSize*(*f*).

Um die Byte-Position innerhalb der Datei zu erhalten, muß der von *FilePos* zurückgelieferte Wert mit der Recordgröße dieser Datei bzw. mit der Größe einer einzelnen Komponente multipliziert werden.

Wenn *FilePos* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-}

den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Grenzen *FilePos* kann nicht auf Textdateien angewendet werden. Diese Funktion setzt voraus, daß die entsprechende Datei offen ist.

Querverweis *FileSize*, *Seek*

FileSize-Funktion

Funktion liefert die Größe einer Datei zurück.

Deklaration `FileSize(var f)`

Ergebnistyp *LongInt*

Erläuterung *FileSize* liefert die Anzahl der Komponenten zurück, die in der angegebenen Datei gespeichert sind. *f* steht für eine typisierte oder untypisierte Datei-Variable. Wenn die Datei leer ist, liefert *FileSize* das Ergebnis 0.

Wenn *FileSize* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus `{$I-}` den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Grenzen *FileSize* kann nicht auf Textdateien angewendet werden. Die Funktion setzt voraus, daß die angegebene Datei offen ist.

Querverweis *FilePos*

Beispiel

```
program FileSizeDemo;
var
  f: file of Byte;
begin
  { Der Name der Datei wird als
    1. Kommandozeilen-Parameter erwartet }
  Assign(f, ParamStr(1));
  Reset(f);
  Writeln('Größe von ', ParamStr(1), ' in Bytes: ',
          FileSize(f));
  Close(f);
end.
```

FillChar-Prozedur

Funktion füllt einen Speicherbereich mit einem bestimmten Wert.

Deklaration `FillChar(var x; count: Word; value)`

Erläuterung *x* steht für einen Variablenbezug beliebigen Typs, der von *FillChar* als Startadresse des zu füllenden Bereichs interpretiert

wird. *count* gibt die Anzahl der mit *value* zu füllenden Speicherzellen an, *value* ist ein Ausdruck des Typs *Char* oder *Byte*.

Achtung: FillChar nimmt keine Bereichsprüfungen vor. Soweit möglich, sollte deshalb der Parameter *count* über *SizeOf* angegeben werden – auf diese Weise sorgt der Compiler dafür, daß *FillChar* nicht versehentlich andere Speicherbereiche mit überschreibt.

FillChar läßt sich nicht zuletzt zum Füllen von Strings verwenden. Bitte beachten Sie, daß das Längenbyte in diesem Fall explizit gesetzt werden muß (siehe »Beispiel«).

Querverweis *Move*

Beispiel

```
var
  s : string[80];
begin   { Setzen eines Strings mit 80 Sternchen }
  FillChar(s, SizeOf(s), '*');
  s[0] := #80;   { Längenbyte setzen }
end.
```

FillEllipse-Prozedur — Graph

Funktion zeichnet einen ausgefüllten elliptischen Kreis.

Deklaration `FillEllipse( X, Y: Integer; XRadius , YRadius: Word)`

Erläuterung *FillEllipse* zeichnet einen elliptischen Kreis um den Mittelpunkt (*X*,*Y*) mit dem Radius *XRadius* (horizontal) und *YRadius* (vertikal). Alle Parameter dieser Funktion können sowohl Variablen als auch Ausdrücke sein; wie bei *Circle*, *Arc* und *Ellipse* wird das Höhen-/Seitenverhältnis des Bildschirms automatisch berücksichtigt.

Nach dem Zeichnen des Kreisbogens (mit der momentan gesetzten Farbe und der durch *SetLineStyle* gesetzten Linienart) füllt *FillEllipse* das Innere des Kreises mit dem durch *SetFillPattern* und *SetFillStyle* vorgegebenen Muster.

Grenzen vor dem Aufruf dieser Prozedur muß ein Grafikmodus gesetzt worden sein.

Querverweis *Arc, Circle, Ellipse, GetAspectRatio, PieSlice*

Beispiel

```
program FillEllipseDemo;
uses Graph;
var
  GraphDriver, GraphMode: Integer;
  x: Integer;

begin
  GraphDriver := Detect;   { automatische Erkennung }
  InitGraph(GraphDriver, GraphMode, '\TP');
```

```
  for x:= 10 downto 1 do
  begin
    SetFillStyle(WideDotFill, x);
    FillEllipse(GetMaxX div 2, GetMaxY div 2,
                200, x * GetMaxY div 20);
  end;

  Readln;
  CloseGraph;
end.
```

FillPoly-Prozedur — Graph

Funktion — zeichnet ein ausgefülltes Polygon.

Deklaration

```
FillPoly(NumPoints: Word; var  PolyPoints)
```

Erläuterung — *PolyPoints* ist ein untypisierter Parameter, der die Koordinaten der Eckpunkte des Polygons enthält. Die Anzahl der Koordinatenpaare wird in *NumPoints* angegeben (einem Ausdruck des Typs *Word*).

Jede Koordinate besteht aus zwei Werten (*x* und *y*) des Typs *Word* und kann über den in *Graph* vordefinierten Datentyp *PointType* angegeben werden:

```
type
  PointType = record
                x, y: Word;
              end;
```

FillPoly beginnt mit dem ersten angegebenen Punkt und zieht von dort aus eine Linie zum zweiten, von dort aus zum dritten usw. Zum Zeichnen einer geschlossenen Figur mit *n* Eckpunkten muß *PolyPoints* *n*+1 Koordinatenpaare enthalten, das letzte Paar (*PolyPoints*(*n*+1)) muß wieder mit dem ersten identisch sein (siehe »Beispiel«).

Zum Zeichnen des Umrisses wird die momentane Zeichenfarbe und die durch *SetLineStyle* gesetzte Linienart verwendet. Nachdem der Umriß gezeichnet ist, berechnet *FillPoly* die dadurch eingeschlossenen Teile zeilenweise (jeweils von links nach rechts) und füllt diese Teile mit dem über *SetFillStyle* bzw. *SetFillPattern* festgelegten Muster.

Wenn beim Zeichnen des Polygons ein Fehler auftritt, dann liefert ein nachfolgender Aufruf von *GraphResult* den Wert -6.

Grenzen — vor dem Aufruf von *FillPoly* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *DrawPoly*, *GetFillSettings*, *GetLineSettings*, *SetColor*, *SetFillStyle*, *SetFillPattern*, *SetLineStyle*, *GraphResult*.

Beispiel

```
program FillPolyDemo;
uses Crt, Graph;

const
  Pentagon: array[1..6] of PointType =
   ( (x: 50;  y: 50),
     (x: 75;  y: 75),
     (x: 62;  y: 100),
     (x: 48;  y: 100),
     (x: 25;  y: 75),
     (x: 50;  y: 50) );

var
  GraphMode, GraphDriver: Integer;

begin
  GraphDriver := Detect;  { autom. Erkennung }
  InitGraph(GraphDriver, GraphMode, '\TP');

  DrawPoly(SizeOf(Pentagon) div SizeOf(PointType),
           Pentagon);
  Readln;
  FillPoly(SizeOf(Pentagon) div SizeOf(PointType),
           Pentagon);
  Readln;
  CloseGraph;
end.
```

FindFirst-Prozedur — Dos

Funktion — sucht ein Verzeichnis nach dem ersten Vorkommen eines Dateinamens ab.

Deklaration

```
FindFirst(Path : string; Attr: Byte; var     S:SearchRec)
```

Erläuterung — *Path* gibt den zu suchenden Dateinamen (z.B. *.*) an und kann optional einen Laufwerksbezeichner und/oder einen Suchweg enthalten. Wenn *Path* nur einen Dateinamen angibt, benutzt *FindFirst* das momentan gesetzte Verzeichnis.

Attr legt die Attribute des zu suchenden Dateieintrags fest. Für diesen Parameter sind die folgenden Konstanten in der Unit *Dos* deklariert:

```
const
  ReadOnly  = $01;
  Hidden    = $02;
  SysFile   = $04;
  VolumeID  = $08;
  Directory = $10;
  Archive   = $20;
  AnyFile   = $3F;
```

Bei der Angabe des Attributs müssen Sie beachten, daß *Findfirst* immer auch Dateien findet, die ein weniger einschränkendes Attribut haben. Wenn Sie z.B. *VolumeID* angeben, »findet« *Findfirst* das Plattenlabel, Unterverzeichnisse und normale Dateien, nicht aber verborgene und System-Dateien.

FindFirst erwartet einen (nicht initialisierten) Parameter des Typs *SearchRec*, über dessen Felder die Ergebnisse der Suche zurückgeliefert werden. Der Typ *SearchRec* ist ebenfalls in der Unit *Dos* definiert (vgl. Kapitel 12 »Standard-Units« des *Programmierhandbuches*):

```
type
  SearchRec = record
                Fill: array[1..21] of Byte;
                Attr: Byte;
                Time: Longint;
                Size: Longint;
                Name: string[12];
              end;
```

Fehler bei der Ausführung von *FindFirst* setzen die globale Variable *DosError*. Mögliche Codes sind 0 (fehlerfrei), 2 (Verzeichnis nicht gefunden) und 18 (Keine weiteren Einträge).

Querverweis *FindNext*, *FExpand*

Beispiel

```
program FindFirstDemo;
uses Dos;

var
  DirInfo: SearchRec;

begin
  FindFirst('*.PAS', Archive, DirInfo);
       { dasselbe wie DIR *.PAS }
  while DosError = 0 do
    begin
      Writeln(DirInfo.Name);
      FindNext(DirInfo);
    end;
end.
```

FindNext-Prozedur — Dos

Funktion setzt eine mit *FindFirst* begonnene Suche fort.

Deklaration `FindNext(var S:SearchRec)`

Erläuterung *FindNext* erwartet eine Variable des Typs *SearchRec*, die durch einen Aufruf von *FindFirst* initialisiert worden sein muß. Jeder

Aufruf von *FindNext* sucht das jeweils nächste Vorkommen eines Dateieintrags, der die durch *FindFirst* festgelegten Bedingungen erfüllt und liefert ihn in der übergebenen Variablen *s* zurück.

Fehler bei der Ausführung von *FindNext* setzen die globale Variable *DosError*: der einzig mögliche Code ist hier 18 (keine weiteren Dateieinträge).

Querverweis *FindFirst*, *FExpand*

Beispiel siehe *FindFirst*

FloodFill-Prozedur Graph

Funktion füllt einen umschlossenen Bereich mit dem momentan gesetzten Füll-Muster.

Deklaration
```
FloodFill( x,y: Integer; Border: Word)
```

Erläuterung *FloodFill* arbeitet im Gegensatz zu *FillPoly* bit-orientiert: ausgehend von dem durch (*x*,*y*) angegebenen Punkt wird der Bildschirm in alle Richtungen »geflutet« (*Flood* = fluten), das Erreichen eines Pixels mit der durch *Border* angegebenen Farbe bricht die Operation in der jeweiligen Richtung ab.

Wenn sich der Punkt (*x*,*y*) innerhalb einer Fläche befindet, die mit Linien der Farbe *Border* umschlossen ist, füllt *FloodFill* diese Fläche. Befindet sich (*x*,*y*) außerhalb dieser Fläche, dann wird der restliche Bildschirm (bzw. das restliche Zeichenfenster) gefüllt, die umschlossene Fläche bleibt frei.

Wenn ein Fehler während der Flächenfüllung auftritt, liefert ein nachfolgender Aufruf von *GraphResult* das Ergebnis -7.

Hinweis: Neben dem Erreichen einer durchgehenden Linie der Farbe *Border* kennt *FloodFill* eine zweite Abbruchbedingung: nach der Ausgabe zweier aufeinanderfolgender *leerer* Zeilen wird die Operation beendet. Diese Grenze macht sich im allgemeinen nur beim Füllen sehr kleiner Flächen mit einem »dünnen« Muster bemerkbar, wie es das folgende Fragment demonstriert:

```
...
SetFillStyle(LtSlashFill, GetMaxColor);
Rectangle(0,0,8,20);   { ein sehr kleines Rechteck }
FloodFill(1,1,GetMaxColor);
...
```

Hier bricht *FloodFill* ab, bevor das Rechteck vollständig ausgefüllt ist. Die Verwendung eines dichteren Füll-Musters (wie z.B. *SlashFill*) beseitigt das Problem.

Grenzen vor der Ausführung von *FloodFill* muß ein Grafikmodus gesetzt worden sein. Um die Kompatibilität mit späteren Versionen

des Compilers zu erhalten, sollte *FillPoly* anstelle von *FloodFill* verwendet werden (soweit möglich).

FloodFill ist nicht zusammen mit dem Treiber des IBM 8514-Adapters (IBM8514.BGI) anwendbar.

Querverweis *GetFillPattern*, *SetFillPattern*, *SetFillStyle*

Beispiel

```
program FloodFillDemo;
uses Graph;
var GraphMode, GraphDriver: Integer;

begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
  SetColor(Red);
  MoveTo (10,10);  LineTo(100,10); { ein rotes Dreieck }
  LineTo(50,100);  LineTo(10,10);
  Readln;
  FloodFill(30,30,Red);  { füllt das Dreieck }
  Readln;
  FloodFill(100,100,Red);{ füllt den restlichen Schirm }
  Readln;
  CloseGraph;
end.
```

Flush-Prozedur

Funktion erzwingt das Schreiben des Dateipuffer-Inhalts einer Textdatei.

Deklaration `Flush(var f: Text)`

Erläuterung *f* steht für eine Textdatei-Variable. Normalerweise haben Ausgaben zu einer mit *Rewrite* oder *Append* geöffneten Textdatei erst dann eine physikalische Schreibaktion zur Folge, wenn der Dateipuffer voll ist. Ein Aufruf von *Flush* erzwingt das physikalische Schreiben des Pufferinhalts unabhängig von seinem »Füllungsgrad« (unter anderem wird diese Prozedur automatisch von *Close* aufgerufen).

Bei Dateien, die mit *Reset* eröffnet wurden (d.h. nur für Leseoperationen), hat *Flush* keine Wirkung.

Wenn *Flush* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus `{$I-}` den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Grenzen die durch *f* bezeichnete Datei muß offen sein.

Querverweis *Append*, *Reset*, *Rewrite*

Frac-Funktion

Funktion liefert den nicht-ganzzahligen Teil des Arguments zurück.

Deklaration `Frac(x: Real)`

Ergebnistyp *Real*

Erläuterung *x* ist ein Real-Ausdruck. Das Ergebnis der Funktion ist der nicht-ganzzahlige Anteil von *x* (d.h. die »Stellen nach dem Komma«) und läßt sich auch über die Formel *x - Int(x)* berechnen.

Querverweis *Int*

Beispiel
```
var
  x : Real;
begin
  x := Frac(123.456);  { x := 0.456 }
end.
```

FreeMem-Prozedur

Funktion gibt einen Speicherbereich bestimmter Größe auf dem Heap wieder frei.

Deklaration `FreeMem(var p: Pointer; size: Word)`

Erläuterung *p* ist eine Zeigervariable beliebigen Typs, der zuvor mit *New*, *GetMem* oder durch eine direkte Zuweisung ein Speicherbereich auf dem Heap zugeordnet wurde. *Size* ist ein Integer-Ausdruck, gibt die Größe dieses Speicherbereichs (bzw. der Variablen) in Bytes an und muß **exakt** mit der Größe des über *p* belegten Speicherbereichs übereinstimmen.

Der Wert von *p* ist nach dem Aufruf von *Dispose* nicht mehr definiert, Zugriffe der Form *p^* stellen einen Fehler dar.

Grenzen wenn *p* nicht auf eine Adresse innerhalb des Heap-Bereichs zeigt, bricht das Programm mit einem Laufzeitfehler ab. *Dispose* und *FreeMem* können nur unter bestimmten Umständen zusammen mit *Mark* und *Release* verwendet werden (siehe Kapitel 14 »Interne Details« des *Programmierhandbuches*).

Querverweis *Dispose*, *GetMem*, *Release*

FSearch-Funktion Dos

Funktion sucht eine Liste von Verzeichnissen nach einem Dateieintrag ab.

Deklaration `FSearch(Path: PathStr; DirList : string)`

Ergebnistyp *PathStr*

Erläuterung *FSearch* sucht die über den String-Ausdruck *Path* angegebene Datei in den als *DirList* angegebenen Verzeichnissen. *DirList* ist ebenfalls ein String-Ausdruck und muß eine Folge von Verzeichnisnamen enthalten, die untereinander (wie in der *Environment*-Variablen PATH) durch Semikolons voneinander getrennt sind.

Die Suche beginnt grundsätzlich in dem Verzeichnis, das momentan als Standard gesetzt ist – falls sich die Datei dort nicht findet, werden die via *DirList* angegebenen Verzeichnisse der Reihe nach abgesucht.

Wenn *FSearch* die Datei in keinem der angegebenen Verzeichnisse lokalisieren kann, liefert die Funktion einen Nullstring zurück; ansonsten ist das Funktionsergebnis der Name der gesuchten Datei zusammen mit dem Suchweg des Verzeichnisses, in dem die Suche erfolgreich war.

Der Datentyp *PathStr* ist in der Unit *Dos* als *string*[79] definiert.

Hinweise: Der Aufruf von *FSearch* mit dem Ergebnis des Funktionsaufrufs *GetEnv('PATH')* als Parameter *DirList* sucht sämtliche Verzeichnisse ab, die momentan als PATH gesetzt sind. Wenn PATH relative Suchwege enthält, läßt sich das Ergebnis von *FSearch* mit *FExpand* zu einem vollständigen Namen erweitern. (Solange PATH nur vollständige Suchwege inklusive Laufwerksbezeichner definiert, liefert *FSearch* bereits ein vollständiges Ergebnis – eine nachfolgende Erweiterung via *FExpand* ist in diesem Fall nicht notwendig).

Querverweis *FExpand*, *FSplit*, *GetEnv*

Beispiel
```
program FSearchDemo;
uses Dos;
var S: PathStr;
begin
  S:= FSearch('TURBO.TPL', GetEnv('PATH'));
  if S = '' then Writeln('TURBO.TPL nicht gefunden!')
  else Writeln('Vollständiger Suchweg `,
               'zu TURBO.TPL: ', S);
end.
```

FSplit-Prozedur Dos

Funktion zerlegt einen vollständigen Dateinamen in seine drei Komponenten »Suchweg«, »Name« und »Suffix«.

Deklaration
```
FSplit(Path: PathStr; var Dir: DirStr;
       var Name: NameStr; var Ext: ExtStr);
```

Erläuterung *FSplit* analysiert den als *Path* angegebenen String-Ausdruck

und ordnet seine Komponenten den als *Dir, Name* und *Ext* übergebenen Stringvariablen zu. Wenn *Path* eine oder mehrere Komponenten nicht enthält, bekommt die entsprechende Variable einen Nullstring zugewiesen.

In *Dir* werden Laufwerksbezeichner und Suchweg (zusammen mit einem Doppelpunkt und den entsprechenden führenden bzw. folgenden umgekehrten Schrägstrichen) gespeichert; *Name* bekommt den Dateinamen zugewiesen, *Ext* das Suffix mit einem führenden Punkt.

Die Datentypen *PathStr, DirStr, NameStr* und *ExtStr* sind in der Unit *Dos* folgendermaßen definiert:

```
type
  PathStr  = string[79];
  DirStr   = string[67];
  NameStr  = string[8];
  ExtStr   = string[4];
```

Hinweis: FSplit liefert eine exakte Unterteilung der als *Path* übergebenen Zeichenfolge, ohne dabei Zeichen zu löschen oder hinzuzufügen. Nach einem Aufruf wie

```
FSplit('C:\TURBO\TEST.PAS', DStr, NStr, EStr);
```

liefert das direkte Verbinden der drei Ergebnisse mit

```
DStr + NStr + EStr
```

also wieder das Ergebnis 'C:\TURBO\TEST.PAS'.

Querverweis *FExpand, FindFirst, FSearch*

Beispiel

```
program FSplitDemo; { Datei FSPLIT.PAS }
uses Dos;

var
  Filename: PathStr;
  DStr: DirStr;
  NStr: NameStr;
  EStr: ExtStr;

begin
   Filename := FExpand('FSPLIT.PAS');
   Writeln(Filename);
   FSplit(Filename, DStr, NStr, EStr);
   Writeln('FSPLIT.PAS ist im Verzeichnis ', DStr,
           ' gespeichert.');
end.
```

GetArcCoords-Prozedur — Graph

Funktion — liefert Daten über den zuletzt ausgeführten Aufruf von *Arc* zurück.

Deklaration

```
GetArcCoords(var ArcCoords: ArcCoordsType)
```

Erläuterung — *GetArcCoords* liefert die Daten des letzten Aufrufs von *Arc* in einer Variablen des Typs *ArcCoordsType* zurück, der in *Graph* folgendermaßen definiert ist:

```
type
  ArcCoordsType = record
    X, Y :       Integer;  { Mittelpunkt }
    Xstart,Ystart,         { Startpunkt }
    Xend, Yend : Integer;  { Endpunkt }
   end;
```

Sinnvoll ist die Verwendung von *GetArcCoords* beispielsweise dann, wenn ein Kreisbogen mit einer Linie oder einem anderen geometrischen Element fortgesetzt werden soll.

Grenzen — vor dem Aufruf von *GetArcCoords* muß ein Grafikmodus gesetzt worden sein. Wenn noch kein Aufruf von *Arc* erfolgt ist, sind die Ergebnisse von *GetArcCoords* undefiniert.

Querverweis — *Arc, Circle, Ellipse, PieSlice, Sector.*

Beispiel

```
program GetArcCoordsDemo;
uses Graph;
var
  GraphMode, GraphDriver: Integer;
  ArcCoords: ArcCoordsType;
begin
  GraphDriver := Detect;  { autom. Erkennung }
  InitGraph(GraphDriver, GraphMode, '\TP');
  Arc(100,100,0,89,30);
  Readln;
  GetArcCoords(ArcCoords);
  MoveTo(0,0);
  with ArcCoords do     { eine Linie zum Mittelpunkt }
    LineTo (X, Y);
  Readln;
  CloseGraph;
end.
```

GetAspectRatio-Prozedur — Graph

Funktion — ermittelt das physikalische Höhen-/Seitenverhältnis des Bildschirms.

Deklaration

```
GetAspectRatio(var  Xasp , Yasp: Word)
```

Erläuterung — jedem Grafiktreiber und jedem Grafikmodus ist ein *Aspekt* zugeordnet, der von Prozeduren wie *Arc, Circle* und *PieSlice* verwendet wird, damit Kreise auch optisch rund erscheinen. Für Quadrate und andere selbstdefinierte Figuren kann dieses Verhältnis über *GetAspectRatio* ermittelt werden. Um beispielsweise ein Quadrat mit einer gegebenen Breite *x* zu zeichnen, wird die Höhe *y* folgendermaßen ermittelt:

```
GetAspectRatio (Xasp, Yasp);
y := Round( x * (Xasp/Yasp));
```

Hinweis: Xasp und *Yasp* beziehen nicht nur die Auflösung des Bildschirms (z.B. 640 * 350 Punkte), sondern auch die physikalischen Dimensionen eines »typischen« Bildschirms (ca. 21 cm Breite und 15 cm Höhe) mit ein.

Mit der Prozedur *SetAspectRatio* läßt sich der vom Grafikpaket verwendete Korrekturfaktor direkt setzen.

Grenzen — vor dem Aufruf von *GetAspectRatio* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *Arc, Circle, GetMaxX, GetMaxY, PieSlice, SetAspectRatio.*

Beispiel

```
program GetAspectRatioDemo;
uses Graph;

var
  GraphMode, GraphDriver: Integer;
  XAsp, YAsp: Word;
  XSeite, YSeite: Word;

begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');

  XSeite := 100;
  { Anpassung der Höhe an den Wert von XSeite, damit
    das Quadrat auch so auf dem Bildschirm erscheint: }
  GetAspectRatio(XAsp, YAsp);
  YSeite := Round((XAsp/YAsp) * XSeite);

  { und Zeichnen des "Quadrats" }
  Rectangle(0, 0, XSeite, YSeite);

  Readln;
  CloseGraph;
end.
```

GetBkColor-Funktion Graph

Funktion liefert die momentan gesetzte Hintergrundfarbe zurück.

Deklaration `GetBkColor`

Ergebnistyp *Word*

Erläuterung *GetBkColor* liefert die Nummer des Eintrags in der Farb-Palette, der momentan als Hintergrundfarbe benutzt wird. Der Wert 0 als Funktionsergebnis bedeutet also, daß Turbo Pascal den ersten Eintrag verwendet (welche Farbe dabei tatsächlich herauskommt, hängt vom Inhalt der Farb-Tabelle ab).

Ein Beispiel: Im Modus CGAC1 enthält die Farb-Palette die vier Einträge *Black*(0), *Red*(1), *Yellow*(2) und *Green*(3). Wenn *GetBkColor* den Wert 2 zurückliefert, wird *Yellow* als Hintergrundfarbe benutzt.

Solange nicht mit *SetBkColor* ein anderer Paletten-Eintrag als Hintergrundfarbe gesetzt wurde, verwendet Turbo Pascal den ersten Eintrag, *GetBkColor* hat also das Ergebnis 0. Abhängig vom verwendeten Grafiktreiber und -modus sind Ergebnisse im Bereich von 0..15 möglich.

Grenzen vor dem Aufruf von *GetBkColor* muß ein Grafikmodus gesetzt worden sein.

Querverweis *GetColor, GetPalette, InitGraph, SetAllPalette, SetBkColor, SetColor, SetPalette.*

Beispiel

```
program GetBkColorDemo;
uses
   Crt, Graph;
var
   GraphMode, GraphDriver: Integer;
   Color: Word;
   Palette: PaletteType;

begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
  GetPalette(Palette); Randomize;
  repeat
    Color := Succ(GetBkColor);
    Delay(100);
    if Color >= Palette.Size
      then Color := 0;
    SetBkColor(Color);
    LineTo(Random(GetMaxX), Random(GetMaxY));
  until KeyPressed;
  Readln;
```

```
  CloseGraph;
end.
```

GetCBreak-Prozedur — Dos

Funktion — ermittelt, bei welchen Operationen DOS auf Ctrl Break prüft.

Deklaration — `GetCBreak(var Break: Boolean)`

Erläuterung — *GetCBreak* ermittelt, ob DOS nur bei Ein-/Ausgaben oder bei jedem Aufruf auf die Eingabe von Ctrl Break prüft und verwendet dazu die Funktion $33 von MS-DOS. Im ersten Fall wird die als *Break* übergebene Variable auf *True* gesetzt, ansonsten bekommt sie den Wert *False*.

Querverweis — *SetCBreak*

GetColor-Funktion — Graph

Funktion — liefert die momentan gesetzte Zeichenfarbe zurück.

Deklaration — `GetColor`

Ergebnistyp — *Word*

Erläuterung — *GetColor* liefert die Nummer des Eintrags in der Farb-Palette, der momentan als Zeichenfarbe benutzt wird. Der Wert 0 als Funktionsergebnis bedeutet also, daß Turbo Pascal den ersten Eintrag verwendet (welche Farbe dabei tatsächlich herauskommt, hängt vom Inhalt der Farb-Tabelle ab).

Ein Beispiel: Im Modus CGAC1 enthält die Farb-Palette die vier Einträge *Black*(0), *Red*(1), *Yellow*(2) und *Green*(3). Wenn *GetColor* den Wert 1 zurückliefert, wird *Red* als Zeichenfarbe benutzt.

Solange nicht mit *SetColor* ein anderer Paletten-Eintrag als Zeichenfarbe gesetzt wurde, verwendet Turbo Pascal den höchsten Eintrag: *GetColor* hat also im Modus CGAC1 das Ergebnis 3. Abhängig vom verwendeten Grafiktreiber und -modus sind Ergebnisse im Bereich von 0..15 möglich.

Grenzen — vor dem Aufruf von *GetColor* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *GetBkColor*, *GetPalette*, *InitGraph*, *SetAllPalette*, *SetColor*, *SetColor*, *SetPalette*.

Beispiel

```
program GetColorDemo;
uses Crt, Graph;

var
  GraphMode, GraphDriver: Integer;
```

```
  Color: Word;
  Palette: PaletteType;

begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
  GetPalette(Palette);
  Randomize;
  repeat
    Color := Succ(GetColor);
    Delay(100);
    if Color >= Palette.Size then
       Color := 0;
    SetColor(Color);
    LineTo(Random(GetMaxX), Random(GetMaxY));
  until KeyPressed;

  Readln;
  CloseGraph;
end.
```

GetDate-Prozedur — Dos

Funktion: ermittelt das momentan gesetzte Kalenderdatum.

Deklaration: `GetDate(var Year , Month, Day, DayofWeek: Word)`

Erläuterung: *GetDate* liefert die Werte für das Kalenderdatum zurück, die von DOS benutzt werden. *Year* hat einen möglichen Wertebereich von 1980..2099, *Month* von 1..12, *Day* von 1..31 und *DayofWeek* (Wochentag) von 0..6, wobei der Wert 0 für Sonntag steht.

Querverweis: *SetDate*, *GetTime*, *SetTime*

GetDefaultPalette-Prozedur — Graph

Funktion: liefert die Farb-Palette zurück, die bei der Initialisierung des Grafiktreibers gesetzt war.

Deklaration: `GetDefaultPalette(var Pal: Palettetype)`

Erläuterung: jeder Grafiktreiber enthält eine Farb-Palette, die bei seinem Start (mit *InitGraph* bzw. *SetGraphMode*) gesetzt wird. Über *GetDefaultPalette* lassen sich die in dieser Standard-Palette gespeicherten Werte jederzeit ermitteln – also auch *nach* Aufrufen von *SetPalette* oder *SetAllPalette*.

Der Datentyp *Palettetype* ist im Unit *Graph* folgendermaßen definiert:

```
const
  MaxColors = 15;

type
  PaletteType = record
     Size  : Byte;
     Colors: array[0..MaxColors] of ShortInt;
     end;
```

Grenzen vor dem Aufruf von *GetDefaultPalette* muß ein Grafikmodus gesetzt worden sein.

Querverweis *InitGraph, GetPalette, SetPalette, SetAllPalette.*

Beispiel

```
program GetDefaultPaletteDemo;

uses Graph;

var
  GraphDriver, GraphMode: Integer;
  x : Integer;
  OrgPal: PaletteType;

begin
  GraphDriver := Detect;   { automatische Erkennung }
  InitGraph(GraphDriver, GraphMode, '\TP');

  for x := 1 to GetPaletteSize-1 do  { Textausgabe in }
  begin                          { verschiedenen Farben }
    SetColor(x);
    OutTextXY(20, (x+1) * 10, 'Irgendein Text');
  end;

{ die Farben etwas durcheinanderbringen }
  for x:= 1 to 100 do
    SetPalette(Random(GetPaletteSize),
               Random(GetPaletteSize));
  Readln;

  GetDefaultPalette(OrgPal);  { Orginal-Palette lesen }
  SetAllPalette(OrgPal);      { und wieder setzen }

  Readln;
  CloseGraph;
end.
```

GetDir-Prozedur

Funktion ermittelt das momentan gesetzte Verzeichnis eines Laufwerks.

Deklaration
```
GetDir(d: Byte; var  s: string)
```

Erläuterung *d* ist ein Ausdruck des Typs *Byte* und gibt das Laufwerk an (0 = momentan gesetztes Laufwerk, 1 = Laufwerk A:, 2 = B: usw); *s* ist eine String-Variable und bekommt von *GetDir* den Namen bzw. Suchweg des Verzeichnisses zugewiesen, der für das entsprechende Laufwerk als Standard gesetzt ist.

Wenn *GetDir* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus `{$I-}` den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Querverweis *ChDir, MkDir, RmDir*

GetDriverName-Funktion Graph

Funktion liefert den Namen des momentan aktiven Grafiktreibers als String.

Deklaration
```
GetDriverName
```

Ergebnistyp *String*

Erläuterung die Funktion liefert den Namen des Grafiktreibers zurück, der durch *InitGraph* aktiviert wurde.

Grenzen vor dem Aufruf von *GetDriverName* muß ein Grafikmodus gesetzt worden sein.

Querverweis *GetModeName, InitGraph*.

Beispiel
```
program GetDriverNameDemo;
uses Graph;

var
  GraphDriver, GraphMode: Integer;

begin
  GraphDriver := Detect;   { automatische Erkennung }
  InitGraph(GraphDriver, GraphMode, '\TP');

  OutText('Aktiver Treiber: '+ GetDriverName);
  Readln;
  CloseGraph;
end.
```

GetEnv-Funktion — Dos

Funktion — liest einen Eintrag aus der Tabelle *Environment*.

Deklaration
```
GetEnv(EnvVar : string)
```

Ergebnistyp — *String*

Erläuterung — die von DOS unterhaltene Tabelle *Environment* besteht aus einer (unsortierten) Folge von Einträgen der Form

```
Name=Zeichenfolge
```

Die »bekanntesten« Einträge dürften COMSPEC (Suchweg zu COMMAND.COM) und PATH (Suchwege für ausführbare Programme) sein.

GetEnv sucht die Tabelle nach dem als *EnvVar* angegebenen String-Ausdruck (Name) ab und liefert den String (Zeichenfolge) zurück, der zusammen mit diesem Namen in der Tabelle gespeichert ist. Für einen *Environment*-Eintrag wie

```
DUMMY=EIN TEXT
```

hat der Aufruf *GetEnv('Dummy')* das Ergebnis 'EIN TEXT' – das vorangestellte Gleichheitszeichen wird von dieser Funktion nicht mit zurückgeliefert.

GetEnv übersetzt (wie der DOS-Befehl SET) die als *EnvVar* angegebene Zeichenfolge automatisch in Großbuchstaben; wenn sich der gesuchte Eintrag nicht finden läßt, ist das Funktionsergebnis ein Nullstring.

Querverweis — *EnvCount, EnvStr*

Beispiel
```
program GetEnvDemo;
uses Dos;
var Line: string;
begin
  Line := GetEnv('COMSPEC');
  if Line = '' then Writeln('Nanu?')
   else Writeln('Suchweg zu COMMAND.COM: ', Line);
  Line := GetEnv('PATH');
  if Line = '' then Writeln('Kein PATH gesetzt!')
   else begin
     Writeln('PATH enthält die folgenden Suchwege:');
     Writeln(Line);
    end;
end.
```

GetFAttr-Prozedur — Dos

Funktion — liefert die Attribute einer Datei zurück.

Deklaration

```
GetFAttr(var f; var Attr: Word)
```

Erläuterung — *f* steht für eine Datei-Variable beliebigen Typs, der mit *Assign* eine externe Diskettendatei zugeordnet worden ist. *GetFAttr* ermittelt die Attribute dieser Datei, die *nicht* offen sein darf, und setzt die übergebene Variable *Attr* entsprechend.

Die einzelnen Attribute lassen sich durch **and**-Operationen mit den folgenden (in der Unit *Dos* definierten) Konstanten ermitteln:

```
const
  ReadOnly  = $01;
  Hidden    = $02;
  SysFile   = $04;
  VolumeID  = $08;
  Directory = $10;
  Archive   = $20;
  AnyFile   = $3F;
```

Fehler während der Ausführung von *GetFAttr* setzen die globale Variable *DosError*, mögliche Codes sind 2 (Datei nicht gefunden), 3 (ungültiger Suchweg) und 5 (Dateizugriff verweigert).

Grenzen — die durch *f* bezeichnete Datei darf nicht offen sein.

Querverweis — *SetFAttr*, *GetFTime*, *SetFTime*

Beispiel

```
program GetFAttrDemo;
uses Dos;
var
  f : file;
  attr: Word;
begin
  { Dateiname wird als
    1. Kommandozeilen-Parameter erwartet }
  Assign(f, ParamStr(1));
  GetFAttr(f, attr);
  Writeln('Datei: ', ParamStr(1));
  if DosError <> 0 then
    Writeln ('DOS-Fehler: ', DosError)
   else begin
    Write ('Attribute: ');
   if attr and ReadOnly <> 0 then
      Write ('schreibgeschützt  ');
   if attr and Hidden <> 0 then Write ('"hidden"  ');
   if attr and SysFile <> 0 then Write ('System  ');
```

```
  if attr and VolumeID <> 0 then Write ('Vol-ID  ');
  if attr and Directory <> 0 then Write ('(Sub)Dir  ');
  if attr and Archive <> 0 then
     Write ('Archive (normale Datei)');
   Writeln;
  end; { else }
end.
```

GetFillPattern-Prozedur — Graph

Funktion — ermittelt das zuletzt mit *SetFillPattern* bzw. *SetFillStyle* gesetzte Bitmuster für Flächenfüllungen.

Deklaration

```
GetFillPattern(var FillPattern: FillPatternType)
```

Erläuterung — im Gegensatz zu *GetFillStyle* liefert *GetFillPattern* nicht die Kennziffer des Musters für Flächenfüllungen zurück, sondern das eigentliche Bitmuster selbst. Der Datentyp *FillPatternType* ist in *Graph* folgendermaßen definiert:

```
type
  FillPatternType = array[1..8] of Byte;
```

Jedes Byte in diesem Array legt acht nebeneinanderliegende Punkte in einer Bildschirmzeile fest, die Bytes selbst werden in aufeinanderfolgende Zeilen ausgegeben.

Hinweis: Wenn seit dem Start des Grafikpakets *via InitGraph* noch kein Füll-Muster (mit *SetFillStyle* bzw. *SetFillPattern*) gesetzt wurde, setzt *GetFillPattern* sämtliche Bytes der als *FillPattern* übergebenen Variablen auf den Wert $FF (was dem vordefinierten Muster *SolidFill* entspricht).

Grenzen — vor dem Aufruf von *GetFillPattern* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *FillPoly*, *GetFillStyle*, *SetFillStyle*, *SetFillPattern*.

Beispiel

```
var
  FillBits: FillPatternType;

begin
  ....
  SetFillStyle(XHatchFill, Green);
  GetFillPattern(FillBits);  { ermittelt die Bitfolge
                               dieses Füll-Musters }
  ....
end.
```

GetFillSettings-Prozedur Graph

Funktion liefert Daten über das momentan gesetzte Füll-Muster und die verwendete Farbe.

Deklaration
```
GetFillSettings(var FillInfo: FillSettingsType)
```

Erläuterung *GetFillSettings* ermittelt die über *SetFillStyle* bzw. *SetFillPattern* gesetzten Parameter und liefert sie in der als *FillInfo* übergebenen Variablen zurück. Der Datentyp *FillSettingsType* ist in *Graph* folgendermaßen definiert:

```
type
  FillSettingsType = record
    Pattern: Word;
    Color  : Word;
  end;
```

Das Feld *Pattern* enthält die Nummer des Bitmusters, mit dem Flächenfüllungen ausgeführt werden (d.h. einen Wert im Bereich von 0..12), *Color* enthält die dazu verwendete Farbe (d.h. einen Wert im Bereich von 0..15).

Grenzen vor dem Aufruf von *GetFillSettings* muß ein Grafikmodus gesetzt worden sein.

Querverweis *FillPoly*, *GetFillPattern*, *SetFillStyle*, *SetFillPattern*.

Beispiel
```
var
  FillInfo: FillSettingsType;
  CurPattern,   { momentanes Muster & Farbe }
  CurColor : Word;

begin
  ....
  GetFillSettings(FillInfo);
  CurPattern := FillInfo.Pattern;  { Nummer des
                                     Füll-Musters }
  CurColor := FillInfo.Color;      { Füll-Farbe }
  ....
end.
```

GetFTime-Prozedur Dos

Funktion ermittelt Datum und Uhrzeit der letzten Veränderung einer Datei.

Deklaration
```
GetFTime(var f; var Time: LongInt)
```

Erläuterung *f* ist eine Datei-Variable beliebigen Typs, die für eine externe und offene Diskettendatei steht. Der über die Variable *Time* zurückgelieferte Wert enthält Datum und Uhrzeit in einem

»gepackten« Format und kann mit der Prozedur *UnpackTime* interpretiert werden.

Fehler bei der Ausführung von *GetFTime* setzen die globale Variable *DosError*; der einzig mögliche Code ist 6 (ungültiger Handle/Datei nicht offen).

Grenzen die durch *f* bezeichnete Datei muß offen sein.

Querverweis *SetFTime*, *PackTime*, *UnpackTime*

GetGraphMode-Funktion — Graph

Funktion ermittelt den momentan gesetzten Grafikmodus und liefert einen entsprechenden Wert zurück.

Deklaration `GetGraphMode`

Ergebnistyp *Integer*

Erläuterung das Funktionsergebnis von *GetGraphMode* liegt grundsätzlich im Bereich von 0..5 – welcher Grafikmodus dabei (mit *InitGraph* oder *SetGraphMode*) tatsächlich gesetzt wurde, hängt vom verwendeten Treiber ab.

In der Unit *Graph* sind die folgenden Konstanten für Grafikmodi definiert:

Name	Wert	Beschreibung
Grafiktreiber CGA		
CGAC0	0	320*200, Palette C0, 4 Farben, 1 Seite
CGAC1	1	320*200, Palette C1, 4 Farben, 1 Seite
CGAC2	2	320*200, Palette C2, 4 Farben, 1 Seite
CGAC3	3	320*200, Palette C3, 4 Farben, 1 Seite
CGAHI	4	640*200, 2 Farben, 1 Seite
Grafiktreiber MCGA		
MCGAC0	0	320*200, Palette C0, 4 Farben, 1 Seite
MCGAC1	1	320*200, Palette C1, 4 Farben, 1 Seite
MCGAC2	2	320*200, Palette C2, 4 Farben, 1 Seite
MCGAC3	3	320*200, Palette C3, 4 Farben, 1 Seite
MCGAMed	4	640*200, 2 Farben, 1 Seite
MCGAHi	5	640*480, 2 Farben, 1 Seite
Grafiktreiber EGAVGA, EGA mit 256 KByte RAM		
EGALo	0	640*200, 16 Farben, 4 Seiten
EGAHi	1	640*350, 16 Farben, 2 Seiten
EGAMonoHi	3	640*350, 2 Farben, 2 Seiten
Grafiktreiber EGAVGA, EGA mit 64 KByte RAM		
EGA64Lo	0	640*200, 16 Farben, 1 Seite
EGA64Hi	1	640*350, 4 Farben, 1 Seite
EGAMonoHi	3	640*350, 2 Farben, 1 Seite

Name	Wert	Beschreibung
Grafiktreiber EGAVGA, VGA		
VGALo	0	640*200, 16 Farben, 2 Seiten
VGAMed	1	640*350, 16 Farben, 2 Seiten
VGAHi	2	640*480, 16 Farben, 1 Seite
Grafiktreiber HERC		
HercMonoHi	0	720*348, 2 Farben, 2 Seiten
Grafiktreiber PC3270		
PC3270Hi	0	720*350, 2 Farben, 1 Seite
Grafiktreiber ATT400		
ATT400C0	0	320*200, Palette C0, 4 Farben, 1 Seite
ATT400C1	1	320*200, Palette C1, 4 Farben, 1 Seite
ATT400C3	2	320*200, Palette C2, 4 Farben, 1 Seite
ATT400C3	3	320*200, Palette C3, 4 Farben, 1 Seite
ATT400Med	4	640*200, 2 Farben, 1 Seite
ATT400Hi	5	640*400, 2 Farben, 1 Seite
Grafiktreiber IBM8514		
IBM8514Lo	0	640*480, 256 Farben, 1 Seite
IBM8514Hi	1	1024*768, 16 Farben, 1 Seite

Mit *GetModeRange* und *GetMaxMode* läßt sich ermitteln, welche Grafikmodi für einen bestimmten Treiber zulässig sind.

Grenzen vor dem Aufruf von *GetGraphMode* muß ein Grafikmodus gesetzt worden sein.

Querverweis *DetectGraph*, *GetMaxMode*, *GetModeRange*, *InitGraph*, *SetGraphMode*.

Beispiel
```
var
  Mode: Integer;
begin
  ....
    Mode := GetGraphMode; { ermittelt gesetzten Modus }
  ....
end.
```

GetImage-Prozedur — Graph

Funktion kopiert einen rechteckigen Bildausschnitt in eine Puffervariable.

Deklaration
```
GetImage( x1, y1, x2, y2: Integer; var    BitMap)
```

Erläuterung *x1*, *y1*, *x2* und *y2* sind Ausdrücke des Typs *Integer* und legen die obere linke bzw. untere rechte Ecke des zu kopierenden Bildausschnitts fest. *BitMap* ist ein untypisierter Parameter, den *GetImage* als Startadresse des Speicherbereichs interpretiert, in den die Bilddaten hineinkopiert werden sollen.

Die ersten beiden 16-Bit-Worte des Ziel-Speicherbereichs von *GetImage* (d.h. der als *BitMap* übergebenen Variablen) werden zur Speicherung der Breite und Höhe des kopierten Ausschnitts verwendet:

```
Breite := x2 - x1 + 1;
Hoehe:= y2 - y1 + 1;
```

BitMap muß also insgesamt vier Bytes mehr Speicherplatz bieten, als die eigentlichen Bilddaten belegen. Die Funktion *ImageSize,* mit der sich der Platzbedarf für einen Bildausschnitt ermitteln läßt, bezieht diese vier Bytes automatisch mit ein.

Grenzen vor dem Aufruf von *GetImage* muß ein Grafikmodus gesetzt worden sein. Die Größe eines Bildausschnitts ist auf 64 KByte Speicherplatzbedarf begrenzt.

Querverweis *ImageSize, PutImage.*

Beispiel

```
program GetImageDemo;
uses Graph;
var  GraphMode, GraphDriver: Integer;
     p : Pointer; Size, MaxX, x: Word;
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
  MaxX := GetMaxX div 4; { ein Viertel des Bildschirms }
  for x := 1 to 100 do begin
   SetColor(Random(16)); { ein paar bunte Linien }
   LineTo(Random(MaxX), Random(GetMaxY));
   end;
  Readln;
  Size := ImageSize(0,0,MaxX,GetMaxY); { Größe
                                       bestimmen }
  GetMem(p, Size);                { Platz auf dem Heap }
  GetImage(0,0,MaxX,GetMaxY,p^);
  for x:= 1 to 3 do
    PutImage(x * MaxX, 0, p^, NormalPut); { 3 Kopien }
  Readln;
  CloseGraph;
end.
```

GetIntVec-Prozedur — Dos

Funktion ermittelt den Inhalt eines Interrupt-Vektors.

Deklaration `GetIntVec(IntNo: Byte; var  Vector: Pointer)`

Erläuterung *IntNo* ist ein Ausdruck des Typs *Byte* und gibt die Nummer des zu ermittelnden Interrupt-Vektors (0..255) an, dessen Inhalt

von *GetIntVec* in der als *Vector* übergebenen Zeigervariablen gespeichert wird.

Nach einem Aufruf wie

```
GetIntVec(0, Zeiger_var);
```

zeigt die Variable *Zeiger_var,* also auf die Startadresse der Routine, die durch den Prozessorbefehl *INT 0* aktiviert wird.

Querverweis *SetIntVec*

GetLineSettings-Prozedur Graph

Funktion ermittelt die durch den letzten Aufruf von *SetLineStyle* gesetzten Parameter.

Deklaration
```
GetLineSettings(var  LineInfo: LineSettingsType)
```

Erläuterung *GetLineSettings* ermittelt die momentan gesetzte Linienart sowie das verwendete Muster und die Liniendicke und speichert die Ergebnisse in der als *LineInfo* übergebenen Variablen.

Der Typ *LineSettingsType* ist in der Unit *Graph* folgendermaßen definiert:

```
type
  LineSettingsType = record
      LineStyle : Word;
      Pattern   : Word;
      Thickness : Word;
    end;
```

Für Linienarten, -muster und -dicken definiert *Graph* die folgenden Konstanten:

```
const
  { Linienarten und -breiten für Get/SetLineStyle }
  SolidLn    = 0;      { durchgezogen }
  DottedLn   = 1;      { gepunktet }
  CenterLn   = 2;      { Punkt Strich Punkt }
  DashedLn   = 3;      { gestrichelt }
  UserBitLn  = 4;      { benutzerdefiniert }

  NormWidth  = 1;      { normale Breite }
  ThickWidth = 3;      { dick }
```

Grenzen vor dem Aufruf von *GetLineSettings* muß ein Grafikmodus gesetzt worden sein.

Querverweis *DrawPoly, SetLineStyle.*

Beispiel
```
var
  LineInfo : LineSettingsType;
  Linienart, Linienmuster, Liniendicke : Word;
```

```
begin
  ....
  GetLineSettings(LineInfo);
  Linienart := LineInfo.LineStyle;
  Linienmuster := LineInfo.Pattern;
  Liniendicke := LineInfo.Thickness;
  ....
end.
```

GetMaxColor-Funktion — Graph

Funktion	liefert die maximal momentan mögliche »Farbnummer«.
Deklaration	`GetMaxColor`
Ergebnistyp	*Word*
Erläuterung	*GetMaxColor* liefert die maximal mögliche Farbnummer, die für den momentan gesetzten Grafikmodus und -treiber verwendet werden kann. Der zurückgelieferte Wert entspricht der Größe der Farb-Palette minus 1 (CGA: 3, EGA: 15 usw).
Grenzen	vor dem Aufruf von *GetMaxColor* muß ein Grafikmodus gesetzt worden sein.
Querverweis	*SetColor*, *GetMaxMode*.

GetMaxMode-Funktion — Graph

Funktion	liefert die Nummer des »höchsten« Grafikmodus für den momentan aktiven Treiber.
Deklaration	`GetMaxMode`
Ergebnistyp	*Word*
Erläuterung	*GetMaxMode* ermittelt den maximal möglichen Wert für *GraphMode* (der normalerweise für den Grafikmodus mit der höchsten Auflösung steht), durch einen direkten Aufruf des momentan geladenen Treibers. Diese Funktion ist im Gegensatz zu *GetModeRange* auch für Grafiktreiber verwendbar, die nicht direkt von Borland erstellt wurden. *Hinweis:* Der niedrigste Grafikmodus für Treiber von Drittanbietern hat per Definition immer die Nummer 0.
Grenzen	vor dem Aufruf von *GetMaxMode* muß ein Grafikmodus gesetzt worden sein.
Querverweis	*GetMaxColor*, *GetModeRange*, *SetGraphMode*.

Beispiel

```
program GetMaxModeDemo;
uses Graph;

var
  GraphDriver, GraphMode: Integer;
  x: Integer;

begin
  GraphDriver := Detect;   { automatische Erkennung }
  InitGraph(GraphDriver, GraphMode, '\TP');
  for x:= 0 to GetMaxMode do   { Namen der Grafikmodi }
    OutTextXY(10, 10* Succ(x), GetModeName(x));

  Readln;
  CloseGraph;
end.
```

GetMaxX-Funktion — Graph

Funktion — ermittelt die maximal mögliche X-Koordinate des Bildschirms.

Deklaration — `GetMaxX`

Ergebnistyp — *Integer*

Erläuterung — *GetMaxX* liefert die maximal mögliche X-Koordinate des Bildschirms zurück, d.h. die X-Koordinate des rechten Bildschirmrandes. Das Ergebnis hängt vom verwendeten Grafiktreiber und -modus ab. Mit einem *Color Graphics Adapter* im Modus CGAC1 (320 * 200 Pixel) liefert *GetMaxX* das Ergebnis 319, mit einer EGA-Karte im Modus EGAHi (640 * 350 Pixel) das Ergebnis 639.

GetMaxX und *GetMaxY* stellen die einzige Möglichkeit dar, ein Grafik-Programm mehr oder weniger unabhängig von der verwendeten Hardware-Ausrüstung zu schreiben. Diese beiden Funktionen sollten grundsätzlich anstelle von Konstanten verwendet werden.

Grenzen — vor dem Aufruf von *GetMaxX* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *GetMaxY, GetX, GetY, MoveTo.*

Beispiel

```
program GetMaxXYDemo;
uses Graph;
var  GraphMode, GraphDriver: Integer;
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');

  { ein Rahmen für den gesamten Bildschirm }
  Rectangle(0, 0,GetMaxX, GetMaxY);
```

```
  Readln;
  CloseGraph;
end.
```

GetMaxY-Funktion — Graph

Funktion	ermittelt die maximal mögliche Y-Koordinate des Bildschirms.
Deklaration	`GetMaxY`
Ergebnistyp	*Word*
Erläuterung	*GetMaxY* liefert die maximal mögliche Y-Koordinate des Bildschirms zurück, d.h. die Y-Koordinate des unteren Bildschirmrandes. Das Ergebnis hängt vom verwendeten Grafiktreiber und -modus ab. Mit einem *Color Graphics Adapter* im Modus CGAC1 liefert *GetMaxY* das Ergebnis 199, mit einer EGA-Karte im Modus EGAHi (640 * 350 Pixel) das Ergebnis 349. *GetMaxY* und *GetMaxY* stellen die einzige Möglichkeit dar, ein Grafik-Programm mehr oder weniger unabhängig von der verwendeten Hardware-Ausrüstung zu schreiben. Diese beiden Funktionen sollten grundsätzlich anstelle von Konstanten verwendet werden.
Grenzen	vor dem Aufruf von *GetMaxY* muß ein Grafikmodus gesetzt worden sein.
Querverweis	*GetMaxY*, *GetX*, *GetY*, *MoveTo*.
Beispiel	siehe *GetMaxX*

GetMem-Prozedur

Funktion	belegt einen Bereich bestimmter Größe auf dem Heap.
Deklaration	`GetMem(var p: Pointer; Size: Word)`
Erläuterung	*GetMem* belegt einen Bereich von *Size* Bytes auf dem Heap und weist der als *p* übergebenen Zeigervariablen die Startadresse dieses Bereichs zu, erzeugt also eine dynamische Variable, auf die über *p^* zugegriffen werden kann. *p* ist eine Zeigervariable beliebigen Typs, *Size* ist ein Ausdruck des Typs *Word*. Wenn der Heap nicht mehr genügend Platz für die Belegung von *Size* Bytes (in einem zusammenhängenden Block) bietet, bricht die Programmausführung mit einem Laufzeitfehler ab. (Kapitel 14 »Interne Details« des *Programmierhandbuches* enthält eine Beschreibung der zugrundeliegenden Mechanismen und der Funktion *HeapFunc*, über die derartige Fehler behandelt werden können).
Grenzen	maximal können mit einem Aufruf von *GetMem* 65521 Bytes

(=64 KByte-$0F) belegt werden. Solange der Heap (noch) nicht fragmentiert ist, reservieren aufeinanderfolgende Aufrufe von *GetMem* einander benachbarte Speicherbereiche. *GetMem* und *FreeMem* können nur unter bestimmten Umständen zusammen mit *Mark* und *Release* verwendet werden (siehe Kapitel 14 »Interne Details« des *Programmierhandbuches*).

Querverweis *FreeMem*, *Dispose*, *New*, *Mark*, *Release*

GetModeName-Funktion — Graph

Funktion liefert den Namen eines der Grafikmodi zurück.

Deklaration
```
GetModeName(ModeNumber: Integer)
```

Ergebnistyp *String*

Erläuterung *GetModeName* erwartet die Nummer eines Grafikmodus als Parameter und liefert den Namen dieses Grafikmodus als String. Ein Beispiel dazu:

```
...
GraphDriver := EGA; GraphMode := EGAHi;
InitGraph(GraphDriver, GraphMode, '\TP');
OutText(GetModeName(GraphMode)); { -> '640 x 350 EGA' }
```

Die Namen der Grafikmodi sind Bestandteil des jeweiligen Treibers – das Ergebnis von *GetModeName* hängt also davon ab, welcher Treiber zum Zeitpunkt des Aufrufs aktiv ist.

Grenzen vor dem Aufruf von *GetModeName* muß ein Grafikmodus gesetzt worden sein (wobei es allerdings keine Rolle spielt, ob der als *ModeNumber* übergebene Parameter denselben Wert wie die Kennziffer dieses Grafikmodus hat oder davon abweicht).

Querverweis *GetDriverName*, *GetGraphMode*, *GetModeRange*.

Beispiel siehe *GetMaxMode*

GetModeRange-Prozedur — Graph

Funktion ermittelt die möglichen Grafikmodi für einen Grafiktreiber.

Deklaration
```
GetModeRange(GraphDriver: Integer;
             var LoMode. HiMode: Integer);
```

Erläuterung *GetModeRange* liefert den niedrigstmöglichen Grafikmodus für *GraphDriver* in *LoMode* und den höchstmöglichen Modus in *HiMode* zurück. Wenn *GraphDriver* einen Wert außerhalb des erlaubten Bereichs enthält (vgl. *InitGraph* bzw. *InstallUserDriver*), dann werden *LoMode* und *HiMode* auf -1 gesetzt.

Hinweis: GetModeRange kann auch dann aufgerufen werden,

wenn kein Grafikmodus gesetzt ist; der als *GraphMode* übergebene Wert muß nicht mit der Kennziffer eines eventuell geladenen Grafiktreibers übereinstimmen.

Querverweis *GetDriverName, DetectGraph, GetGraphMode, GetMaxMode.*

Beispiel

```
program GetModeRangeDemo;
uses Graph;
var
  MinMode, MaxMode: Integer;
begin
  GetModeRange(EGA64, MinMode, MaxMode);
  Writeln('Niedrigster Grafikmodus für EGA64: ',
           MinMode);
  Writeln('Höchster Grafikmodus für EGA64: ', MaxMode);
end.
```

GetPalette-Prozedur Graph

Funktion liefert Informationen über die momentan gesetzte Farb-Palette.

Deklaration `GetPalette(var Palette: PaletteType)`

Erläuterung *GetPalette* liefert die einzelnen Einträge der momentan gesetzten Farb-Palette und die Gesamtgröße der Palette in der als *Palette* übergebenen Variablen zurück. Der Datentyp *PaletteType* ist in der Unit *Graph* folgendermaßen definiert:

```
const
  MaxColors = 15;
type
  PaletteType = record
      Size   : Byte;
      Colors : array[0..MaxColors] of ShortInt;
    end;
```

Das Feld *Size* enthält die Anzahl der tatsächlich benutzten Einträge des Arrays *Colors*, sein Wert ist vom aktiven Grafikmodus und dem verwendeten Treiber abhängig (CGA: vier Farben, EGA: 16 Farben usw). Die Elemente von *Colors* (*0..Size-1*) enthalten die einzelnen Einträge der aktiven Farb-Palette.

Grenzen vor dem Aufruf von *GetPalette* muß ein Grafikmodus gesetzt worden sein.

Querverweis *GetDefaultPalette, GetMaxColor, GetPaletteSize, SetPalette.*

Beispiel

```
program GetPaletteDemo;
uses Graph;
var GraphMode, GraphDriver: Integer;
```

```
    Color: Word;
    Palette: PaletteType;
begin
  GraphDriver := Detect; { automatische Erkennung }
  InitGraph(GraphDriver, GraphMode, '\TP');
  GetPalette(Palette);
  for Color := 0 to Palette.Size-1 do
  begin
    SetColor(Color);
    Line(0, Color * 10, GetMaxX, Color * 10);
   end;
  Readln;
  CloseGraph;
end.
```

GetPaletteSize-Funktion — Graph

Funktion	liefert die Größe der momentan gesetzten Farb-Palette zurück (d.h. die Anzahl der Einträge).
Deklaration	`GetPaletteSize`
Ergebnistyp	*Integer*
Erläuterung	mit *GetPaletteSize* läßt sich ermitteln, wieviele Einträge die Farb-Palette im momentan gesetzten Grafikmodus enthält (z.B. 4 für eine CGA-Karte in einem der Farbmodi; 16 für eine EGA-Karte im Modus *EGAHi*).
Grenzen	vor dem Aufruf von *GetPaletteSize* muß ein Grafikmodus gesetzt worden sein.
Querverweis	*GetDefaultPalette*, *GetMaxColor*, *GetPalette*, *SetPalette*.

GetPixel-Funktion — Graph

Funktion	liefert die Farbe eines Pixels zurück.
Deklaration	`GetPixel( X,Y: Integer)`
Ergebnistyp	*Word*
Erläuterung	*GetPixel* liefert die Farbe des Pixels mit den Koordinaten (*X,Y*) zurück. Ermittelt wird die *Farbnummer* – in welcher physikalischen Farbe das Pixel tatsächlich auf dem Bildschirm erscheint, hängt vom entsprechenden Eintrag der Farb-Palette ab.
Grenzen	vor dem Aufruf von *GetPixel* muß ein Grafik-Modus gesetzt worden sein.
Querverweis	*PutPixel*, *GetImage*, *PutImage*, *SetWriteMode*.

Beispiel

```
var  Color: Word;
begin
  ....
  Color := GetPixel(0,89);
  ....
end.
```

GetTextSettings-Prozedur — Graph

Funktion — liefert Informationen über die Parameter für folgende Ausgaben mit *OutText* und *OutTextXY*.

Deklaration

```
GetTextSettings(var  TextInfo: TextSettingsType)
```

Erläuterung — *GetTextSettings* ermittelt die mit *SetTextStyle* und *SetTextJustify* gesetzten Parameter und liefert sie in der als *TextInfo* übergebenen Variablen zurück. Der Datentyp *TextSettingsType* und die möglichen Werte seiner Felder sind in der Unit *Graph* folgendermaßen definiert:

```
type
  TextSettingsType = record
      Font      : Word;
      Direction : Word;
      CharSize  : Word; { Werte von 1..10 }
      Horiz     : Word;
      Vert      : Word;
    end;

const
  DefaultFont   = 0;   { 8x8 Bit pixelweise definiert }
  TriplexFont   = 1;   { Vektor-Zeichensätze }
  SmallFont     = 2;
  SansSerifFont = 3;
  GothicFont    = 4;
  HorizDir   = 0;      { von links nach rechts }
  VertDir    = 1;      { von unten nach oben }

  NormSize   = 1;      { normale Textgröße }
```

Grenzen — vor dem Aufruf von *GetTextSettings* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *InitGraph*, *SetTextJustify*, *SetTextStyle*, *TextHeight*, *TextWidth*.

Beispiel

```
var
  Schriftart, Richtung, Groesse, Hoehe, Breite: Word;
  TextInfo: TextSettingsType;
  GraphDriver, GraphMode: Integer;
begin
```

```
  GraphDriver := Detect;  { automatische Erkennung }
  InitGraph(GraphDriver, GraphMode, '\TP');

  GetTextSettings(TextInfo);
  with TextInfo do
  begin
    Schriftart := Font;    { siehe oben (Konstanten) }
    Richtung := Direction;  { Schreibrichtung }
    Groesse := CharSize;  { Vergrößerungsfaktor }
    Breite := Horiz;    { "Standard"-Breite }
    Hoehe := Vert;   { Höhe eines Zeichens }
  end;
   { .... }
end.
```

GetTime-Prozedur — Dos

Funktion — liefert die Uhrzeit des Systems zurück.

Deklaration — `GetTime(var Hour, Min, Second, Sec100: Word)`

Erläuterung — *GetTime* setzt die übergebenen Variablen mit den Daten des Betriebssystems. Mögliche Werte für *Hour* sind 0..23, für *Minute* und *Second* 0..59, für *Sec100* (hundertstel Sekunden) 0..99.

Querverweis — *GetDate, SetDate, GetTime, UnpackTime*

GetVerify-Prozedur — Dos

Funktion — ermittelt den Stand des DOS-Flags *Verify*.

Deklaration — `GetVerify(var Verify: Boolean)`

Erläuterung — über das Flag *Verify* wird festgelegt, ob DOS geschriebene Diskettensektoren automatisch überprüft oder nicht. (Normalerweise ist dieses Flag auf *False* gesetzt – eine direkte Überprüfung findet also nicht statt).

GetVerify kopiert das Flag von *Dos* in die als *Verify* übergebene Variable: *True* bedeutet, daß DOS alle Diskettensektoren nach dem Schreiben noch einmal liest und auf diese Weise Prüfsummen verifiziert.

Querverweis — *SetVerify*

GetViewSettings-Prozedur — Graph

Funktion — liefert Informationen über das momentan gesetzte Zeichenfenster.

Deklaration — `GetViewSettings(var Viewport: ViewPortType)`

Erläuterung *GetViewSettings* liefert die Grenzen des Zeichenfensters und einen boolschen Wert für die Clip-Funktion (*True* = aktiv) in der als *Viewport* übergebenen Variablen zurück. Der Datentyp *ViewPortType* ist in der Unit *Graph* folgendermaßen definiert:

```
type
  ViewPortType = record
    x1, y1, x2, y2 : Integer; { obere linke & untere
                                rechte Ecke des
                                Fensters }
    Clip           : Boolean; { True = Clipping aktiv }
  end;
```

Grenzen vor dem Aufruf von *GetViewSettings* muß ein Grafikmodus gesetzt worden sein.

Querverweis *ClearViewPort, SetViewPort.*

Beispiel

```
var  ViewPort : ViewPortType;
begin
  ....
  GetViewSettings(ViewPort);
  if ViewPort.Clip then
     OutText('Clip-Funktion ist aktiv.');
   else  OutText('Clip-Funktion ist nicht aktiv.');
   ....
end.
```

GetX-Funktion — Graph

Funktion liefert die X-Koordinate der momentanen Position des Grafik-Cursors zurück.

Deklaration `GetX`

Ergebnistyp *Integer*

Erläuterung der zurückgelieferte Wert ist relativ zum momentan gesetzten Zeichenfenster. Die absolute X-Koordinate ergibt sich durch die Addition der linken Fenstergrenze, die sich mit *GetViewSettings* ermitteln läßt (siehe vorangehender Abschnitt).

Der Grafik-Cursor wird durch die Prozeduren *SetViewPort*, *MoveTo* und die meisten Zeichenaktionen bewegt. In einer Befehlsfolge wie

```
SetViewPort(0, 0, GetMaxX, GetMaxY, ClipOn);
MoveTo(5,5);
SetViewPort(10, 10, 100, 100, ClipOff);
MoveTo(5,5);
```

setzt der erste Aufruf von *SetViewPort* den Grafik-Cursor auf die

absoluten Koordinaten (*0,0*), der darauffolgende *MoveTo*-Befehl setzt ihn auf die Koordinaten (*5,5*). Die nächsten beiden Befehle setzen ein Zeichenfenster, dessen Ursprung nicht mit der oberen linken Ecke des Bildschirms zusammenfällt: *SetViewPort* setzt den Grafik-Cursor auf die absoluten Koordinaten (*10,10*); beim darauffolgenden *MoveTo*-Befehl wird der relative Bezug zum Fenster deutlich: er setzt den Cursor auf die absoluten Koordinaten (*15,15*).

Grenzen — vor dem Aufruf von *GetX* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *GetViewSettings, GetY, InitGraph, MoveTo, SetViewPort.*

Beispiel

```
program GetXDemo;
uses graph;
var
  GraphMode, GraphDriver: Integer;
  x, y: Integer;
begin
  GraphDriver := Detect;  { automatische Erkennung }
  InitGraph(GraphDriver, GraphMode, '\TP');
  OutText('Hier fangen wir an.');
  x:= GetX; y := GetY;  { Ermitteln der Position }
  OutTextXY(20 ,10, 'Jetzt sind wir dort...');
  OutTextXY(x, y, 'und nun wieder hier!');
  Readln;
  CloseGraph;
end.
```

GetY-Funktion

Funktion — liefert die Y-Koordinate der momentanen Position des Grafik-Cursors zurück.

Deklaration — `GetY`

Ergebnistyp — *Integer*

Erläuterung — der zurückgelieferte Wert ist relativ zum momentan gesetzten Zeichenfenster. Die absolute Y-Koordinate ergibt sich durch die Addition der oberen Fenstergrenze, die sich mit *GetViewSettings* ermitteln läßt (siehe *GetViewSettings*).

Der Grafik-Cursor wird durch die Prozeduren *SetViewPort*, *MoveTo* und das Zeichnen von Linien bewegt. In einer Befehlsfolge wie

```
SetViewPort(0, 0, GetMaxX, GetMaxY, ClipOn);
MoveTo(5,5);
SetViewPort(10, 10, 100, 100, ClipOff);
MoveTo(5,5);
```

setzt der erste Aufruf von *SetViewPort* den Grafik-Cursor auf die absoluten Koordinaten *(0,0)*, der darauffolgende *MoveTo*-Befehl setzt ihn auf die Koordinaten *(5,5)*. Die nächsten beiden Befehle setzen ein Zeichenfenster, dessen Ursprung nicht mit der oberen linken Ecke des Bildschirms zusammenfällt: *SetViewPort* setzt den Grafik-Cursor auf die absoluten Koordinaten *(10,10)*; beim darauffolgenden *MoveTo*-Befehl wird der relative Bezug zum Fenster deutlich: er setzt den Cursor auf die absoluten Koordinaten *(15,15)*.

Grenzen — vor dem Aufruf von *GetY* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *GetViewSettings*, *GetX*, *InitGraph*, *MoveTo*, *SetViewPort*.

Beispiel — siehe *GetX*

GotoXY-Prozedur — Crt

Funktion — setzt den Cursor (im Textbildschirm).

Deklaration

```
GotoXY( X, Y: Byte)
```

Erläuterung — *GotoXY* setzt den (Text-) Cursor auf die durch die Integer-Ausdrücke *X* und *Y* angegebene Spalte bzw. Zeile, wobei (1,1) der oberen linken Ecke des Bildschirms bzw. des momentan gesetzten Textfensters entspricht.

GotoXY arbeitet relativ zu einem gesetzten Textfenster. Eine Befehlsfolge wie

```
Window(20,10,60,20);
GotoXY(1,1);
```

setzt den Cursor in die 20. Spalte der 10. Zeile des Bildschirms (also in die obere linke Ecke des mit *Window* festgelegten Fensters).

Grenzen — wenn auch nur einer der beiden Parameter von *GotoXY* ungültig ist (d.h. außerhalb der momentanen Fenstergrenzen liegt), dann wird der Cursor überhaupt nicht bewegt.

Querverweis — *Window*, *WhereX*, *WhereY*.

GraphDefaults-Prozedur — Crt

Funktion — setzt sämtliche Parameter des Grafikpakets auf die Standardvorgaben zurück.

Deklaration

```
GraphDefaults
```

Erläuterung — *GraphDefaults* setzt die Standardvorgaben für

- das Zeichenfenster (auf den gesamten Bildschirm);

- den Grafik-Cursor (Punkt (0,0));
- den Hintergrund (schwarz), die Zeichenfarbe (*MaxColor*) und die Farb-Palette (*DefaultPalette*);
- Flächenfüllungen (*SolidFill*, *MaxColor*);
- und Textausgaben (*DefaultFont*, *HorizDir*, 1).

Der Inhalt des Bildschirms bleibt unverändert.

Querverweis *InitGraph*.

GraphErrorMsg-Funktion — Graph

Funktion liefert den Text einer Fehlermeldung des Grafikpakets.

Deklaration `GraphErrorMsg(ErrorCode: Integer)`

Ergebnistyp *String*

Erläuterung *GraphErrorMsg* liefert den Text der Fehlermeldung zurück, die zu dem übergebenen Integer-Ausdruck *ErrorCode* gehört. (Als *ErrorCode* wird man normalerweise das Funktionsergebnis von *GraphResult* übergeben – siehe »Beispiel«.

Mit *GraphErrorMsg* kann ein Programm ohne weiteren Aufwand aussagekräftige Meldungen von sich geben (z.B. »Device driver not found« anstelle von »Fehlercode: -3«).

Hinweis: Diese Funktion ist auch dann verfügbar, wenn das Grafikpaket nicht initialisiert worden ist.

Querverweis *GraphResult*, *DetectGraph*, *InitGraph*.

Beispiel

```
var
  GraphDriver, GraphMode: Integer;
  GraphError: Integer;
begin
  GraphDriver := Detect;  { automatische Erkennung }
  InitGraph(GraphDriver, GraphMode, '\TP');
  GraphError := GraphResult;
  if GraphError <> 0 then   { Fehler ? }
     begin
       Writeln('Grafik-Fehler: ',
                GraphErrorMsg(GraphError));
       Write('Weiter mit <RETURN>...');
       Readln;
       Halt;
     end;
  ....
end.
```

GraphResult-Funktion — Graph

Funktion liefert den Fehlerstatus der letzten Grafik-Operation zurück.

Deklaration `GraphResult;`

Ergebnistyp *Integer*

Erläuterung *GraphResult* ist das grafische Gegenstück zu *IOResult* und hat exakt dieselbe Arbeitsweise – nur eben für Grafik-Operationen. Der Fehlerstatus wird nach jedem Aufruf von *GraphResult* automatisch wieder auf 0 gesetzt; für mehrfache Prüfungen müssen Sie das Ergebnis von *GraphResult* deshalb einer Variablen zuweisen, mit der dann die Tests ausgeführt werden.

Die möglichen Ergebnisse von *GraphResult* sind in *Graph* als Konstanten definiert:

Name	Wert	Text/Beschreibung
grOk	0	*No error* fehlerfreie Ausführung
grNoInitGraph	-1	(*BGI*) *graphics not installed* (*use initgraph*) Grafiktreiber (.BGI) nicht geladen *InitGraph* muß zuerst aufgerufen werden
grNotDetected	-2	*Graphics hardware not detected* kein grafikfähiger Adapter vorhanden / gewünschter Modus wird nicht unterstützt
grFileNotFound	-3	*Device driver file not found* Grafiktreiber-Datei nicht gefunden
grInvalidDriver	-4	*Invalid device driver file* Grafiktreiber-Programm defekt (falsche Version, Datei zerstört...)
grNoLoadMem	-5	*Not enough memory to load driver* nicht genug Platz im Hauptspeicher für den Grafiktreiber
grNoScanMem	-6	*Out of memory in scan fill* nicht genug Platz im Hauptspeicher (*Bar, FillPoly, PieSlice, Sector*)
grNoFloodMem	-7	*Out of memory in flood fill* nicht genug Platz im Hauptspeicher (*FloodFill*)
grFontNotFound	-8	*Font file not found* Zeichensatz-Datei nicht gefunden
grNoFontMem	-9	*Not enough memory to load font* nicht genug Platz im Hauptspeicher für den Zeichensatz
grInvalidMode	-10	*Invalid Graphics mode for selected driver* Grafikmodus wird vom geladenen Treiber nicht unterstützt

Name	Wert	Text/Beschreibung
grError	-11	*Generic Error* nicht näher klassifizierbarer Fehler
grIOerror	-12	*Graphics I/O error* Ein-/Ausgabefehler beim Laden von Dateien
grInvalidFont	-13	*Invalid font file* Zeichensatz-Datei ungültig (falsche Version, Datei zerstört...)
grInvalidFontNum	-14	*Invalid font file number* Kennziffer für Zeichensatz nicht definiert

Über die Funktion *GraphErrorMsg* kann zu jedem dieser Werte ein entsprechender Textstring erzeugt werden.

Der durch *GraphResult* ermittelbare Status wird durch die folgenden Routinen gesetzt:

Bar	*ImageSize*	*SetFillPattern*
Bar3D	*InitGraph*	*SetFillStyle*
ClearViewPort	*InstallUserDriver*	*SetGraphBufSize*
CloseGraph	*InstallUserFont*	*SetGraphMode*
DetectGraph	*PieSlice*	*SetLineStyle*
DrawPoly	*RegisterBGIDriver*	*SetPalette*
FillPoly	*RegisterBGIFont*	*SetTextJustify*
FloodFill	*SetAllPalette*	*SetTextStyle*
GetGraphMode		

Querverweis *GraphErrorMsg, IOResult.*

Beispiel siehe *GraphErrorMsg*

Halt-Prozedur

Funktion bricht die Ausführung eines Programms ab.

Deklaration
```
Halt [(Exitcode: Word)]
```

Erläuterung *Halt* bricht die Ausführung des Programms mit sofortiger Wirkung ab und führt einen Rücksprung zur Kommandoebene von DOS bzw. zu dem Prozeß aus, von dem aus das Programm gestartet wurde.

Exitcode ist ein (optionaler) Ausdruck des Typs *Word* und kann benutzt werden, um einen Exit-Code zu übergeben, der sich mit *DosExitCode* innerhalb des aufrufenden Programms bzw. dem Befehl ERRORLEVEL innerhalb einer Batch-Datei ermitteln läßt. *Halt* ohne Angabe dieses Parameters entspricht dem Aufruf *Halt(0)*.

Hinweis: Wie in Kapitel 14 »Interne Details« des *Program-*

mierhandbuches beschrieben, werden via *ExitProc* festgelegte Prozeduren auch bei einem Abbruch mit *Halt* ordnungsgemäß durchgeführt.

Querverweis *Exit*, *RunError*

Hi-Funktion

Funktion liefert das höherwertige Byte des Arguments zurück.

Deklaration `Hi(x)`

Ergebnistyp *Byte*

Erläuterung *x* ist ein Ausdruck mit dem Typ *Integer* oder *Word*. Das Funktionsergebnis ist das höherwertige Byte des Wertes von *x* und ist vorzeichenlos.

Querverweis *Lo*, *Swap*

Beispiel
```
var  w: Word;
begin
  w := Hi($1234);   { w := $12; }
end.
```

HighVideo-Prozedur Crt

Funktion setzt »hohe Intensität« für folgende Ausgaben von Zeichen.

Deklaration `HighVideo`

Erläuterung die Unit *Crt* definiert eine globale Variable namens *TextAttr*, die das jeweils verwendete Attribut für Textausgaben enthält. *HighVideo* setzt das höchstwertige Bit für die Vordergrundfarbe in dieser Variablen (und sorgt damit dafür, daß die Auswahl der Textfarben 0..7 effektiv die Farben 8..15 benutzt).

Querverweis *NormVideo*, *LowVideo*, *TextColor*, *TextBackground*.

Beispiel
```
uses Crt;
begin
  TextAttr := LightGray; Writeln('Hellgrau');
  HighVideo; Writeln('Weiß');   { Hellgrau -> Weiß }
end.
```

ImageSize-Funktion Graph

Funktion berechnet die Größe einer Puffervariablen zur Speicherung von Bildausschnitten.

Deklaration `ImageSize( x1, y1, x2, y2: Integer)`

Ergebnistyp *Word*

Erläuterung *ImageSize* berechnet die Anzahl von Bytes, die zur Speicherung eines Bildausschnitts mit *GetImage* benötigt werden (inklusive der vier »Extra-Bytes« für Höhe und Breite sowie eines weiteren reservierten Wortes mit 16 Bit). *x1*, *y1*, *x2* und *y2* sind Ausdrücke des Typs *Integer* und legen die obere linke bzw. untere rechte Ecke des Bildausschnitts fest.

Die Anzahl der Bytes, die zur Speicherung dieses Ausschnitts notwendig sind, ist vom jeweiligen Grafikmodus abhängig und ergibt sich aus den folgenden Gleichungen:

```
Breite := x2 - x2 + 1;
Hoehe := y2 - y1 + 1;
Pixelzahl := Breite * Hoehe;
Bytezahl := (Pixelzahl * Bits_Pixel) div 8;
```

(*Bits_Pixel* steht für die Anzahl von Bits, die zur Speicherung eines einzelnen Pixels erforderlich sind, und hat beispielsweise im Modus CGAC1 den Wert 2, im Monochrom-Modus den Wert 1, und im Modus EGA16 den Wert 4.)

Grenzen vor dem Aufruf von *ImageSize* muß ein Grafikmodus gesetzt worden sein. Wenn der Platzbedarf des Bildausschnitts die Grenze von 64 KByte übersteigt, liefert *ImageSize* den Wert 0; ein folgender Aufruf von *GraphResult* hat das Ergebnis -11 (*grError*).

Querverweis *GetImage*, *PutImage*.

Beispiel siehe *GetImage*

Inc-Prozedur

Funktion erhöht eine Variable.

Deklaration `Inc(x [, n: LongInt])`

Erläuterung *x* ist eine Variable ordinalen Typs, der optionale Parameter *n* ist ein Integer-Ausdruck. Wenn *n* nicht angegeben ist, erhöht *Inc* die Variable *x* um 1 (und entspricht damit der Zuweisung `x := x+1`), ansonsten wird *x* um *n* erhöht (`x := x+n`). *Inc* ist eine Art **inline**-Prozedur und erzeugt optimierten Code. Die Vorteile dieser »Abkürzung« kommen am meisten in engen Schleifen und bei komplexen Adressierungen zum Tragen.

Querverweis *Dec*, *Pred*, *Succ*

Beispiel

```
var
  IntVar : Integer;
  LongVar: LongInt;
begin
```

```
  IntVar := 10; LongVar := 23948723;
  Inc(IntVar);          { IntVar := IntVar + 1 }
  Inc(LongVar,5);       { LongVar := LongVar + 5);
end.
```

InitGraph-Prozedur — Graph

Funktion — initialisiert das Grafikpaket und setzt einen Grafikmodus.

Deklaration

```
InitGraph(var GraphDriver: Integer,
          var GraphMode: Integer, PathtoDriver: string)
```

Erläuterung

GraphDriver und *GraphMode* sind **var**-Parameter. Wenn *GraphDriver* den Wert 0 hat (= Konstante *Detect*), muß *GraphMode* nicht initialisiert sein: *InitGraph* ruft zuerst sämtliche via *InstallUserDriver* verzeichneten »Detect«-Routinen und dann die Prozedur *DetectGraph* auf, um einen der Hardware-Ausrüstung entsprechenden Treiber auszuwählen und einen dazu passenden Grafikmodus festzusetzen. (Das Laden des Treibers geschieht wieder durch *InitGraph*, die Umschaltung in den Grafikmodus und die Initialisierung sämtlicher Parameter durch die Routine *GraphDefaults*).

Wenn *GraphDriver* einen Wert ungleich 0 hat, dann wird dieser Parameter als Nummer des zu ladenden Grafiktreibers interpretiert. In diesem Fall muß auch *GraphMode* initialisiert sein (ein Aufruf von *DetectGraph* findet hier nicht statt).

Der String-Ausdruck *PathtoDriver* legt einen Suchweg für die Grafiktreiber (.BGI-Dateien) fest; wenn hier ein Nullstring angegeben ist, sucht *InitGraph* im momentan gesetzten Standardverzeichnis.

InitGraph reserviert für den Grafiktreiber (via *GraphGetMem*) einen entsprechenden Speicherplatz auf dem Heap, der beim Beenden des Grafikpakets (mit *CloseGraph* via *GraphFreeMem*) wieder freigegeben wird. Kapitel 14 »Interne Details« des *Programmierhandbuches* beschreibt die zugrundeliegenden Mechanismen und mögliche Eingriffe über selbstdefinierte Routinen zur dynamischen Verwaltung des Speichers.

Der Aufruf von *InitGraph* setzt die als *GraphDriver* und *GraphMode* übergebenen Variablen auf die Nummer des geladenen Grafiktreibers bzw. den gesetzten Grafikmodus. Wenn sich bei der Initialisierung des Grafikpakets ein Fehler ergibt, enthält *GraphDriver* danach einen Wert, den auch ein Aufruf der Funktion *GraphResult* liefern würde:

-2	keinen grafikfähigen Video-Adapter gefunden
-3	Grafiktreiber-Datei (.BGI) nicht gefunden
-4	Grafiktreiber ungültig/Datei zerstört
-5	nicht genug Speicherplatz zum Laden des Treibers
-10	Grafikmodus für gewählten Treiber nicht definiert

InitGraph setzt sämtliche Parameter des Grafik-Pakets auf Standardwerte (Grafik-Cursor, Palette, Zeichenfarbe und -fenster etc).

Die folgenden Konstanten für Grafiktreiber sind in der Unit *Graph* definiert:

```
const  Detect   = 0;    { automatische Erkennung }
       CGA      = 1;
       MCGA     = 2;
       EGA      = 3;
       EGA64    = 4;
       EGAMono  = 5;
       IBM8514  = 6;
       HercMono = 7;
       ATT400   = 8;
       VGA      = 9;
       PC3270   = 10;
```

Dazu kommen die Konstanten für die Grafikmodi der einzelnen Treiber:

Name	Wert	Beschreibung
Grafiktreiber CGA		
CGAC0	0	320*200, Palette C0, 4 Farben, 1 Seite
CGAC1	1	320*200, Palette C1, 4 Farben, 1 Seite
CGAC2	2	320*200, Palette C2, 4 Farben, 1 Seite
CGAC3	3	320*200, Palette C3, 4 Farben, 1 Seite
CGAHI	4	640*200, 2 Farben, 1 Seite
Grafiktreiber MCGA		
MCGAC0	0	320*200, Palette C0, 4 Farben, 1 Seite
MCGAC1	1	320*200, Palette C1, 4 Farben, 1 Seite
MCGAC2	2	320*200, Palette C2, 4 Farben, 1 Seite
MCGAC3	3	320*200, Palette C3, 4 Farben, 1 Seite
MCGAMed	4	640*200, 2 Farben, 1 Seite
MCGAHi	5	640*480, 2 Farben, 1 Seite
Grafiktreiber EGAVGA, EGA mit 256 KByte RAM		
EGALo	0	640*200, 16 Farben, 4 Seiten
EGAHi	1	640*350, 16 Farben, 2 Seiten
EGAMonoHi	3	640*350, 2 Farben, 2 Seiten
Grafiktreiber EGAVGA, EGA mit 64 KByte RAM		
EGA64Lo	0	640*200, 16 Farben, 1 Seite
EGA64Hi	1	640*350, 4 Farben, 1 Seite
EGAMonoHi	3	640*350, 2 Farben, 1 Seite
Grafiktreiber EGAVGA, VGA		
VGALo	0	640*200, 16 Farben, 2 Seiten
VGAMed	1	640*350, 16 Farben, 2 Seiten
VGAHi	2	640*480, 16 Farben, 1 Seite

Name	Wert	Beschreibung
Grafiktreiber HERC		
HercMonoHi	0	720*348, 2 Farben, 2 Seiten
Grafiktreiber PC3270		
PC3270Hi	0	720*350, 2 Farben, 1 Seite
Grafiktreiber ATT400		
ATT400C0	0	320*200, Palette C0, 4 Farben, 1 Seite
ATT400C1	1	320*200, Palette C1, 4 Farben, 1 Seite
ATT400C3	2	320*200, Palette C2, 4 Farben, 1 Seite
ATT400C3	3	320*200, Palette C3, 4 Farben, 1 Seite
ATT400Med	4	640*200, 2 Farben, 1 Seite
ATT400Hi	5	640*400, 2 Farben, 1 Seite
Grafiktreiber IBM8514		
IBM8514Lo	0	640*480, 256 Farben, 1 Seite
IBM8514Hi	1	1024*768, 16 Farben, 1 Seite

Mit *GetModeRange* und *GetMaxMode* läßt sich ermitteln, welche Grafikmodi für einen bestimmten Treiber zulässig sind.

Grenzen

- Der Heap des Programms muß zumindest Platz für den Grafiktreiber bieten (rund 15 KByte).
- Die Grafikkarte des Zenith Z-449 verhält sich wie eine VGA-Karte – folglich setzen *DetectGraph* und *InitGraph* hier den Modus mit 640 * 480 Punkten. Falls der angeschlossene Monitor damit überfordert ist, müssen die Werte für *GraphDriver* und *GraphMode* vor dem Aufruf von *InitGraph* explizit für einen Modus mit niedrigerer Auflösung gesetzt werden (*GraphDriver = VGA, GraphMode = VGAMed*).
- Der IBM 8514-Adapter verhält sich normalerweise wie eine VGA-Karte und wird von *DetectGraph* bzw. *InitGraph* auch als solche behandelt: die Modi *IBM8514Lo* und *IBM8514Hi* sind nur verfügbar, wenn *GraphDriver* und *GraphMode* vor dem Aufruf von *InitGraph* explizit gesetzt werden (*GraphDriver = IBM8514*, *GraphMode = IBM8514Lo* bzw. *IBM8514Hi*).

Querverweis *CloseGraph*, *DetectGraph*, *GraphDefaults*, *GraphResult*, *InstallUserDriver*, *RestoreCrtMode*, *SetGraphBufSize*, *SetGraphMode*, *RegisterBGIDriver*.

Beispiel

```
program  InitGraphDemo;
uses  Graph;
var
  GraphMode, GraphDriver: Integer;
  GraphError: Integer;
  Heap1, Heap2: LongInt;
```

```
begin
  Heap1:= MemAvail;  { Platz auf dem Heap }
  GraphDriver := Detect;  { automatische Erkennung }
  InitGraph(GraphDriver, GraphMode, '\TP');

  if GraphResult <> 0 then begin
    Writeln('Grafik-Fehler: ', GraphDriver);
    Writeln('<RETURN drücken.... ');
    Readln; Halt;
   end;

  Heap2 := MemAvail;
  CloseGraph;
  Writeln('Verwendeter Grafiktreiber: ', GraphDriver);
  Writeln('Gesetzter Grafikmodus: ', GraphMode);
  Writeln('Größe des Treibers: ', Heap1-Heap2,
          ' Bytes.');
end.
```

Insert-Prozedur

Funktion — fügt einen Stringteil in einen String ein.

Deklaration — `Insert(source : string; var s: string ; index: Integer)`

Erläuterung — *source* ist ein String-Ausdruck, *index* ein Integer-Ausdruck. *s* ist eine String-Variable, in die *Insert* den Inhalt von *source* ab der durch *index* angegebenen Zeichenposition einfügt. Wenn *index* größer als die Länge von *s* ist, hat *Insert* denselben Effekt wie *Concat*.

Insert begrenzt die Länge von *s* automatisch auf 255, überzählige Zeichen werden (ohne Fehlermeldung) abgeschnitten.

Querverweis — *Delete, Copy, Concat, Length, Pos*

Beispiel

```
var
  s : string
begin
  s := 'Das ist artig!';
  Insert('groß', s, 9);  { s := 'Das ist großartig!' }
end.
```

InsLine-Prozedur — Crt

Funktion — fügt eine Leerzeile an der Position des Cursors ein.

Deklaration — `InsLine`

Erläuterung — *InsLine* rollt alle Zeilen zwischen der momentanen Position des Cursors und dem Ende des Bildschirms bzw. Textfensters über

einen BIOS-Aufruf um eine Zeile nach unten; die unterste Zeile verschwindet dabei.

Die Zeile, auf der sich der Cursor befindet, wird mit Leerzeichen (und dem für den Hintergrund gesetzten Attribut) gefüllt, die Position des Cursors im Bildschirm bleibt unverändert.

InsLine arbeitet relativ zu einem gesetzten Textfenster. Eine Befehlsfolge wie

```
Window(20,10,60,20);
InsLine;
```

fügt eine (40 Zeichen breite) Zeile auf der Bildschirmzeile 10 ein, die Zeile 20 wird dabei gelöscht.

Querverweis *DelLine, Window*

InstallUserDriver-Funktion — Graph

Funktion zur Installation von Grafiktreibern, die nicht direkt von Borland angeboten werden.

Deklaration `InstallUserDriver(Name : string; AutoDetectPtr: Pointer)`

Ergebnistyp *Integer*

Erläuterung mit *InstallUserDriver* kann das Grafikpaket um Treiber für spezielle Grafikkarten erweitert werden, die sich nach dieser »Installation« wie die Standard-Treiber via *InitGraph* aktivieren lassen.

Die Funktion erwartet als Parameter *Name* den Namen des Grafiktreibers (z.B. 'HIRES'), der auch unter diesem Namen als *.BGI-Datei auf der Diskette gespeichert sein muß (für unser Beispiel also als HIRES.BGI).

InstallUserDriver prüft die als *Name* angegebene Datei (wobei die Endung .BGI automatisch angehängt wird) und liefert eine Integerzahl als Ergebnis, die von *InitGraph* als Kennziffer für diesen Grafiktreiber verwendet werden kann.

Wenn wir davon ausgehen, daß Sie eine spezielle Grafik-Karte nebst einem dazugehörigen Treiber HIRES.BGI erworben haben, dann sieht die »Installation« dieses Treibers im einfachsten Fall so aus:

```
...
     { Installation der Treiberdatei HIRES.BGI }
GraphDriver = InstallUserDriver ('HIRES', nil);
GraphMode = <gewünschter Grafikmodus dieses Treibers>
InitGraph(GraphDriver,
GraphMode, '\TP');  { Aktivierung }
```

Der Parameter *AutoDetectPtr* kann entweder den Wert **nil** haben oder auf eine Pascal-Routine zeigen. Im zweiten Fall läßt sich die »Installation« eines Grafiktreibers noch etwas weiter generalisieren: Wenn *AutoDetectPtr* einen Wert <> **nil** hat, bindet das Grafikpaket die über diesen Zeiger angegebene Routine in die automatischen Prüfungen von *InitGraph* ein. Nach der Definition einer Funktion wie

```
{$F+}
function DetectHIRES: Integer;   { Prüfung auf }
var FoundHIRES;          { »HIRES-Karte vorhanden« }
begin
  { Prüfung der Hardware des Computers:   }
    ....
    if not FoundHIRES { <- HIRES-Karte nicht gefunden }
       DetectHIRES := grError
     else     { Ansonsten den Standardmodus der Karte }
       DetectHIRES := DefaultMode;    { zurückliefern }
end;
{$F-}
```

und dem Einbinden dieser Routine mit

```
xxx := InstallUserDriver('HIRES', @DetectHIRES);
```

ruft *initgraph* beim Aufruf mit der Konstanten *Detect* – also

```
GraphMode := Detect;
InitGraph(GraphMode, GraphDriver, '\TP');
```

vor allen Standard-Prüfungen – die Routine *DetectHIRES* auf. Diese Routine muß prüfen, ob der Computer mit einer entsprechenden Karte ausgerüstet ist – wenn ja, sollte sie den »Standardmodus« dieser Karte zurückliefern; wenn nein, den Fehlercode *grError*. Im ersten Fall aktiviert *InitGraph* in unserem Beispiel den Treiber HIRES mit dem Grafikmodus, den die »Detect-Routine« zurückgeliefert hat; im zweiten Fall wird die Hardware des Computers als nächstes auf das Vorhandensein einer Standard-Karte (CGA, EGA, Hercules usw.) geprüft.

Achtung: Die »Detect-Routine« *muß* als **far** deklariert sein!

Mit *InstallUserDriver* können maximal zehn zusätzliche Grafiktreiber installiert werden, deren »Detect-Routinen« *InitGraph* in der Reihenfolge der Installation (und vor allen eingebauten Prüfroutinen) aufruft.

Querverweis *InitGraph*, *InstallUserFont*, *RegisterBGIDriver*.

InstallUserFont-Funktion — Graph

Funktion zur Installation von Vektor-Zeichensätzen, die nicht direkt von Borland angeboten werden.

Deklaration `InstallUserFont(FontFileName : string)`

Ergebnistyp *Integer*

Erläuterung mit *InstallUserFont* kann das Grafikpaket um Vektor-Zeichensätze erweitert werden, die sich nach dieser »Installation« wie die Standard-Zeichensätze via *SetTextStyle* aktivieren lassen.

Die Funktion erwartet als Parameter *FontFileName* den Namen des Zeichensatzes (z.B. 'FUNNY'), der auch unter diesem Namen als *.CHR-Datei auf der Diskette gespeichert sein muß (für unser Beispiel also als FUNNY.CHR).

InstallUserFont prüft die als *FontFileName* angegebene Datei (wobei die Endung .CHR automatisch angehängt wird) und liefert einen Integer als Ergebnis, der von *SetTextStyle* als Kennziffer für diesen Zeichensatz verwendet werden kann.

Mit *InstallUserFont* können maximal zwanzig zusätzliche Zeichensätze in einem Programm installiert werden. Wenn die Zeichensatz-Tabelle keinen weiteren Eintrag mehr aufnehmen kann, liefert die Funktion das Ergebnis *DefaultFont* (=0), ein nachfolgender Aufruf von *GraphResult* hat das Ergebnis *grError*.

Querverweis *InstallUserDriver*, *RegisterBGIFont*, *SetTextStyle*.

Int-Funktion

Funktion liefert den ganzzahligen Anteil des Arguments zurück.

Deklaration `Int(x:Real)`

Ergebnistyp *Real*

Erläuterung *x* ist ein Ausdruck des Typs *Real*. Das Funktionsergebnis ist der ganzzahlige Anteil von *x*.

Querverweis *Frac*, *Trunc*, *Round*

Beispiel
```
var
  r : Real;
begin
  r := Int(123.456);   { r := 123.0 }
end.
```

Intr-Prozedur Dos

Funktion führt einen Software-Interrupt aus.

Deklaration `Intr(IntNo: Byte; var RegsRegisters)`

Erläuterung *IntNo* ist ein Ausdruck des Typs *Byte* und gibt die Nummer des Interrupts (0..255) an, der ausgeführt werden soll. Mit der als *Regs* übergebenen Variablen werden die Register des Prozessors vor der Ausführung des Interrupt-Befehls gesetzt. Nach der

Ausführung speichert *Intr* die Prozessor-Register wieder in *Regs* zurück. (Diese Variable dient sowohl zur Übergabe von Parametern als auch zum Speichern des Ergebnisses). Der Datentyp *Registers* ist in der Unit *Dos* folgendermaßen definiert:

```
type
  Registers = record
    case Integer of
      0 : (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Word);
      1 : (AL,AH,BL,BH,CL,CH,DL,DH: Byte);
    end;
```

Weitere Details finden Sie in Kapitel 14 »Interne Details« des *Programmierhandbuches*.

Grenzen — Software-Interrupts, die bestimmte Werte der Register SS:SP voraussetzen (oder diese Register verändern), können über *Intr* nicht ausgeführt werden.

Querverweis — *MsDos*

IOResult-Funktion

Funktion — liefert den Fehlerstatus der letzten Ein-/Ausgabeoperation zurück.

Deklaration — `IOResult`

Ergebnistyp — *Integer*

Erläuterung — Im Modus `{$I+}` codiert der Compiler nach jeder Ein- und Ausgabeoperation den Aufruf einer Prüfroutine, die das Programm im Falle eines Fehlers mit einer entsprechenden Meldung abbricht.

Im Modus `{$I-}` unterbleibt diese automatische Prüfung: hier kann (und muß) das Programm mögliche Fehler über den Aufruf von *IOResult* selbst prüfen. Mögliche Ergebnisse (Fehlercodes) sind in Kapitel 3 »Fehlermeldungen und Codes« zusammengefaßt – der Wert 0 steht für »fehlerfreie Ausführung«.

Hinweis: Nach einem Fehler werden folgende Ein- und Ausgabebefehle solange ignoriert, bis das Programm *IOResult* aufgerufen hat.

Jeder Aufruf von *IOResult* setzt die intern gespeicherte Fehlernummer auf 0 zurück. Mehrfache Prüfungen wie

```
if IOResult <> 0
  then if IOResult = 2 then ... { FALSCH! }
```

liefern aus diesem Grund nicht das gewünschte Ergebnis und müssen entweder über eine Hilfsvariable, also beispielsweise mit

```
Fehler := IOResult;
if Fehler <> 0
  then if Fehler = 2 then ....
```

formuliert werden – oder man greift für die erste Prüfung direkt auf die globale Variable *InOutRes* zu:

```
if InOutRes <> 0 then
  case IOResult of
    ...
```

Querverweis *GraphResult*

Beispiel

```
program IOResultDemo;
var
  f : file of Byte;

begin
 { Dateiname wird als
   1. Kommandozeilen-Parameter vorausgesetzt }
  Assign(f, ParamStr(1));
  {$I-} Reset(f); {$I+}
  if (IOResult = 0) then
    Writeln('Dateigröße: ', FileSize(f), ' Bytes.')
   else
    Writeln('Datei nicht gefunden.');
end.
```

Keep-Prozedur — Dos

Funktion beendet ein Programm und macht es speicherresident.

Deklaration `Keep(ExitCode: Word)`

Erläuterung *Keep* liefert den als *ExitCode* übergebenen *Word*-Ausdruck (wie die Prozedur *Halt*) als Exit-Code des Programms an DOS zurück und sorgt (über einen DOS-Funktionsaufruf) dafür, daß das gesamte Programm im Speicher verbleibt.

Achtung! Keep macht das *gesamte* Programm speicherresident – inklusive Daten-Segment, Stack und Heap-Bereich. Bei der Compilierung muß also auf jeden Fall eine Obergrenze des Heaps festgelegt werden – ansonsten betrachtet DOS den Hauptspeicher als »komplett belegt« und das System stürzt ab, weil nicht einmal mehr COMMAND.COM geladen werden kann.

Grenzen speicherresidente Programme setzen eine sehr gute Kenntnis der internen Struktur des Computers voraus – wir verweisen auf die Dokumentation von DOS.

Querverweis *DosExitCode*

KeyPressed-Funktion Crt

Funktion prüft, ob eine Taste gedrückt wurde.

Deklaration `KeyPressed`

Ergebnistyp *Boolean*

Erläuterung *KeyPressed* prüft, ob der Tastaturpuffer »noch nicht abgeholte« Zeichen enthält. Wenn ja, ist das Funktionsergebnis *True*. Die Prüfung erstreckt sich nur auf Tastendrücke, die auch lesbare Zeichen erzeugen (Shift, Ctrl, NumLock usw. ohne die gleichzeitige Eingabe eines »richtigen« Zeichens werden also nicht als Tastendruck behandelt).

Hinweis: KeyPressed nimmt nur eine Prüfung vor, ohne eventuell eingegebene Zeichen »abzuholen«. Wenn also ein Zeichen eingegeben wurde, dann liefern aufeinanderfolgende Aufrufe von *KeyPressed* so lange das Ergebnis *True*, bis das Zeichen (z.B. mit *ReadKey*) auch tatsächlich gelesen wird.

Querverweis *ReadKey*

Beispiel

```
uses Crt;
begin
  repeat
   Write('Xx');      { füllt den Bildschirm }
  until KeyPressed;  { bis zu einem Tastendruck }
end.
```

Length-Funktion

Funktion liefert die dynamische Länge eines Strings zurück.

Deklaration `Length( s: string)`

Ergebnistyp *Integer*

Erläuterung *s* ist ein String-Ausdruck. Das Funktionsergebnis entspricht der Anzahl der in *s* enthaltenen Zeichen.

Querverweis *Concat*, *Copy*, *Delete*, *Insert*, *Pos*

Beispiel

```
var
  s : string;
  i : Integer;
begin
  s := 'abc';
  for i := 1 to Length(s) do
   s[i] := UpCase(s[i]);    { ergibt 'ABC' }
  Writeln(s);
end.
```

Line-Prozedur — Graph

Funktion — zeichnet eine Linie zwischen zwei Punkten.

Deklaration
```
Line(x1,y1,x2,y2: Integer)
```

Erläuterung — (*x1,y1*) stellt den Startpunkt, (*x2,y2*) den Endpunkt der Linie dar; alle vier Parameter sind Integer-Ausdrücke. Zum Zeichnen wird die durch den letzten Aufruf von *SetLineStyle* gesetzte Linienart und die durch *SetColor* festgelegte Farbe verwendet; der jeweils letzte Aufruf von *SetWriteMode* legt fest, ob der Zeichenvorgang direkt oder als XOR-Operation mit dem vorherigen Bildinhalt stattfindet.

Der Grafik-Cursor wird durch *Line* (im Gegensatz zu *LineTo*) nicht bewegt. Das zweite der folgenden Beispiele wird deshalb erst durch den *MoveTo*-Befehl mit dem ersten identisch:

```
{ Beispiel 1 - mit LineTo: }
MoveTo(100,100);
LineTo(200,200); {setzt den Grafik-Cursor automatisch}

{ Beispiel 2 - mit Line: }
Line(100,100,200,200);        {keine Bewegung des GC!}
MoveTo(200,200);
```

Grenzen — vor dem Aufruf von *Line* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *Lineto, MoveTo, Rectangle, SetColor, SetLineStyle, SetWriteMode.*

Beispiel
```
program LineDemo;
uses Crt, Graph;
var
  GraphMode, GraphDriver: Integer;
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');

  Randomize;
  repeat
    Line(Random(200), Random(200),
         Random(200), Random(200));
  until KeyPressed;

  Readln;
  CloseGraph;
end.
```

LineRel-Prozedur — Graph

Funktion — zeichnet eine Linie relativ zur momentanen Position des Grafik-Cursors.

Deklaration
```
LineRel( Dx, Dy: Integer)
```

Erläuterung — *Dx* und *Dy* sind Integer-Ausdrücke und geben die relative Entfernung des Linien-Endpunkts zur momentanen Position des Grafik-Cursors an. Die durch *LineRel* ausgeführte Operation ist identisch mit den folgenden Befehlen:

```
xZiel := GetX + Dx;  { momentane X-Position plus Dx }
yZiel := GetY + Dy;  { momentane Y-Position plus Dy }
LineTo(xZiel, yZiel);
```

LineRel verwendet die durch *SetLineStyle* gesetzte Linienart und die durch *SetColor* festgelegte Zeichenfarbe; der jeweils letzte Aufruf von *SetWriteMode* legt fest, ob der Zeichenvorgang direkt oder als XOR-Operation mit dem vorherigen Bildinhalt stattfindet.

Der Grafik-Cursor wird durch die Zeichenoperation auf den Endpunkt der Linie gesetzt ((*xZiel,yZiel*) im zuvor gezeigten Beispiel).

Relative Operationen sind vor allem dann gut verwendbar, wenn ein Objekt unabhängig von seiner absoluten Position innerhalb des Bildschirms bzw. eines Fensters gezeichnet werden soll.

Grenzen — vor dem Aufruf muß ein Grafikmodus gesetzt worden sein.

Querverweis — *Line, LineTo, MoveRel, MoveTo, SetColor, SetLineStyle, SetWriteMode.*

Beispiel
```
begin
  ....
  MoveTo(1,2);
  LineRel(10,10);  { Linie von (1,2) nach (11,12) }
  ....
end.
```

LineTo-Prozedur — Graph

Funktion — zeichnet eine Linie von der momentanen Position des Grafik-Cursors aus zu einem angegebenen Punkt.

Deklaration
```
LineTo( x, y: Integer)
```

Erläuterung — *LineTo* benutzt die mit *SetLineStyle* gesetzte Linienart und die durch *SetColor* festgelegte Farbe, um von der momentanen Position des Grafik-Cursors aus eine Linie zu dem durch (*x,y*) angegebenen Punkt zu zeichnen (*x* und *y* sind Integer-Aus-

drücke). Ob der Zeichenvorgang direkt oder als XOR-Operation mit den vorherigen Bildinhalt stattfindet, wird durch den jeweils letzten Aufruf von *SetWriteMode* bestimmt.

Nach der Operation steht der Grafik-Cursor auf dem Endpunkt der Linie, d.h. auf den Koordinaten (*x*,*y*). Die Befehlsfolge

```
MoveTo(100,100);
LineTo(200,200);
```

zeichnet eine Linie vom Punkt (100,100) zum Punkt (200,200) und ist identisch mit

```
Line(100,100,200,200);
MoveTo(200,200);
```

Die zweite Befehlsfolge (mit *Line* anstelle von *LineTo*) erzeugt weniger Code und wird schneller ausgeführt – der Grafik-Cursor muß hier allerdings mit einem expliziten Befehl versetzt werden.

Grenzen vor dem Aufruf von *LineTo* muß ein Grafikmodus gesetzt worden sein.

Querverweis *Line*, *LineRel*, *MoveTo*, *MoveRel*, *SetColor*, *SetLineStyle*, *SetWriteMode*.

Beispiel

```
begin
  ....
  Randomize;
  repeat
    LineTo(Random(200), Random(200));
  until KeyPressed;
end.
```

Ln-Funktion

Funktion liefert den natürlichen Logarithmus des Arguments zurück.

Deklaration `Ln(x: Real)`

Ergebnistyp *Real*

Erläuterung *x* ist ein Real-Ausdruck. Das Funktionsergebnis ist der natürliche Logarithmus von *x*, d.h. der Wert, mit dem die Konstante *e* potenziert werden muß, um *x* zu erhalten.

Grenzen Argumente kleiner/gleich 0 sind mathematisch nicht definiert und erzeugen einen Laufzeitfehler.

Querverweis *Exp*

Lo-Funktion

Funktion liefert das niederwertige Byte des Arguments zurück.

Deklaration `Lo(x)`

Ergebnistyp *Byte*

Erläuterung *x* ist ein Ausdruck des Typs *Integer* oder *Word*. Das Funktionsergebnis ist das niederwertige Byte des Wertes von *x* und ist vorzeichenlos.

Querverweis *Hi*, *Swap*

Beispiel
```
var
  w : Word;
begin
  w := Lo($1234);   { w := $34 }
end.
```

LowVideo-Prozedur Crt

Funktion setzt »niedrige Intensität« für folgende Zeichenausgaben.

Deklaration `LowVideo`

Erläuterung die Unit *Crt* deklariert eine globale Variable namens *TextAttr*, die das jeweils aktuelle Attribut für Ausgaben von Textzeichen speichert. *LowVideo* löscht das höchstwertige Bit der Vordergrundfarbe in dieser Variablen – mit dem Effekt, daß die Textfarben 8..15 den Farben 0..7 zugeordnet werden.

Querverweis *HighVideo*, *NormVideo*, *TextColor*, *TextBackground*.

Beispiel
```
uses Crt;
begin
 TextAttr := White; Writeln('Weiß'); { Weiß (Farbe 15) }
 LowVideo; Writeln('Hellgrau');  { Hellgrau (Farbe 7) }
end.
```

Mark-Prozedur

Funktion hält die momentane Spitze des Heaps in einer Zeigervariablen fest.

Deklaration `Mark(var p:Pointer)`

Erläuterung *p* ist eine Zeigervariable beliebigen Typs und bekommt von *Mark* die Adresse zugewiesen, die momentan die Spitze des Heaps darstellt. Durch einen Aufruf von *Release* mit *p* als Argument läßt sich der mit *Mark* festgehaltene Zustand des Heaps wieder herstellen. Anders gesagt: *Release* entfernt sämt-

liche dynamischen Variablen, die nach dem Aufruf von *Mark* erzeugt worden sind.

Grenzen *Mark* und *Release* können nur unter größter Vorsicht abwechselnd mit *Dispose* und *FreeMem* benutzt werden (siehe Kapitel 14 »Interne Details« des *Programmierhandbuches*).

Querverweis *Release*, *Dispose*, *FreeMem*, *GetMem*, *New*

MaxAvail-Funktion

Funktion liefert den Umfang des größten freien Blocks im Heap zurück.

Deklaration `MaxAvail`

Ergebnistyp *LongInt*

Erläuterung *MaxAvail* prüft zuerst, wieviel Platz oberhalb der momentanen Spitze des Heaps noch zur Verfügung steht; falls mit *Dispose*/*FreeMem* gearbeitet wird und eine Fragmentliste vorhanden ist, wird der Heap zusätzlich auf vorhandene Löcher geprüft (siehe Kapitel 14 »Interne Details« des *Programmierhandbuches*).

Zurückgeliefert wird der Umfang des größten zusammenhängenden Blocks. Wenn *MaxAvail* beispielsweise das Ergebnis 30000 hat, kann zumindest noch eine dynamische Variable mit 30000 Byte Umfang erzeugt werden.

Hinweis: Der insgesamt freie Platz auf dem Heap (unabhängig von einer eventuellen Fragmentierung) läßt sich mit der Funktion *MemAvail* ermitteln.

Die Minimalgröße eines Speicherbereichs auf dem Heap beträgt 1 Byte; minimale und maximale Gesamtgröße des Heaps lassen sich über den Compiler-Befehl `{$M}` festlegen (siehe Kapitel 2 »Compiler-Befehle«).

Querverweis *MemAvail*

Beispiel

```
program MaxAvailDemo;
type
  Eintrag = record
     Name : string[30];
     Alter: Byte;
   end;
var
  p : Pointer;

begin
  if MaxAvail < SizeOf(Eintrag) then
    Writeln('Nicht genug Speicherplatz')
   else begin
    { Belegung eines Bereichs auf dem Heap }
    GetMem(p, SizeOf(Eintrag));
```

```
    { ... }
  end;
end.
```

MemAvail-Funktion

Funktion — liefert die Gesamtmenge des noch freien Speicherplatzes auf dem Heap zurück.

Deklaration — `MemAvail`

Ergebnistyp — *LongInt*

Erläuterung — *MemAvail* addiert die Größen sämtlicher Löcher des Heaps sowie den freien Speicherplatz oberhalb der momentanen Heap-Spitze auf und liefert das Ergebnis als Gesamtmenge des freien Speicherplatzes zurück.

Hinweis: Sobald ein Programm die Prozeduren *Dispose* und/oder *FreeMem* verwendet, entstehen nicht zusammenhängende Löcher im Heap (siehe Kapitel 14 »Interne Details« des *Programmierhandbuches*). Aus diesem Grund ist es mehr als unwahrscheinlich, daß der von *MemAvail* angegebene freie Speicherplatz tatsächlich uneingeschränkt zur Verfügung steht.

Über die Funktion *MaxAvail* läßt sich bestimmen, welchen Umfang der größte zusammenhängende Block hat, d.h. wieviele Bytes durch einzelne Aufrufe von *New* oder *GetMem* maximal belegt werden können.

Die Minimalgröße eines Speicherbereichs auf dem Heap beträgt 1 Byte; minimale und maximale Gesamtgröße des Heaps lassen sich über den Compiler-Befehl `{$M}` festlegen (siehe Kapitel 2 »Compiler-Befehle«).

Querverweis — *MaxAvail*

Beispiel

```
begin
  Writeln ('Insgesamt ', MemAvail, 'Bytes verfügbar');
  Writeln ('Größter Block: ', MaxAvail, ' Bytes');
end.
```

MkDir-Prozedur

Funktion — erzeugt ein Unterverzeichnis.

Deklaration — `MkDir(s: string)`

Erläuterung — *s* ist ein String-Ausdruck, gibt den Namen des zu erzeugenden Unterverzeichnisses an und kann optional einen Laufwerksbezeichner und/oder einen Suchweg enthalten. Eine Datei oder

ein Unterverzeichnis des in *s* angegebenen Namens bzw. Suchwegs darf noch nicht existieren.

Wenn der durch *s* angegebene Name mit einem umgekehrten Schrägstrich (backslash) beginnt, bezieht er sich auf das Stammverzeichnis des Laufwerks; ansonsten bezieht er sich auf das momentan gesetzte Verzeichnis des aktiven bzw. angegebenen Laufwerks.

Wenn *MkDir* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus `{$I-}` den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Querverweis *RmDir, ChDir, GetDir*

Beispiel

```
program MkDirDemo;
begin
  { Verzeichnisname wird als
    1. Kommandozeilen-Parameter angenommen }
  {$I-} MkDir(ParamStr(1)); {I+}
  if IOResult <> 0 then
    Writeln('Verzeichnis konnte nicht erzeugt werden!')
   else
    Writeln('Directory ', ParamStr(1), 'erzeugt.');
end.
```

Move-Prozedur

Funktion kopiert Bytes von einem Speicherbereich in einen anderen.

Deklaration `Move(var Source, Dest; count: Word)`

Erläuterung *Source* und *Dest* sind Variablenbezüge beliebigen Typs und werden von *Move* als Startadressen des Quellbereichs (*source*) bzw. Zielbereichs (*dest*) interpretiert.

count ist ein Ausdruck des Typs *Word* und gibt an, wieviele Bytes von *Source* nach *Dest* kopiert werden sollen.

Achtung: Move nimmt keine Prüfung des Zielbereichs vor. Wo immer möglich, sollte deshalb die Anzahl der zu kopierenden Bytes über *SizeOf* bestimmt werden.

Befinden sich Ziel- und Quellbereich innerhalb desselben Segments, sorgt *Move* automatisch für die richtige Richtung des Kopiervorgangs (Kopie mit aufsteigenden /absteigenden Adressen). Da dynamischen Variablen durch die Normalisierung immer eine eigene Segmentadresse zugewiesen wird, kann es auf dem Heap (außer nach direkten Eingriffen des Programmierers) keine Überlappungen geben, *Move* arbeitet daher bei unterschiedlichen Segmentadressen ohne weitere Prüfungen.

Querverweis *FillChar*

Beispiel

```
var
  a : array[1..4] of Byte;
  b : LongInt;
  x : Integer;
begin
  b := $23847699;
  Move(b, a, SizeOf(b));
  Writeln('Die einzelnen Bytes der Variablen haben ',
          'folgende Werte: ');
  for x:= 1 to 4 do
   Write(a[x]:4);
  Writeln;
end.
```

MoveRel-Prozedur — Graph

Funktion — bewegt den Grafik-Cursor relativ zu seiner momentanen Position, ohne zu zeichnen.

Deklaration

```
MoveRel( Dx, Dy: Integer)
```

Erläuterung — *MoveRel* bewegt den Grafik-Cursor um den durch die Integer-Ausdrücke *Dx* und *Dy* angegebenen Betrag relativ zu seiner vorherigen Position. Derselbe Effekt läßt sich auch durch die folgenden Befehle erreichen (vgl. *LineRel*):

```
xZiel := GetX + Dx;  { momentane X-Position plus Dx }
yZiel := GetY + Dy;  { momentane Y-Position plus Dy }
MoveTo(xZiel, yZiel);
```

Relative Bewegungen sind vor allem dann sinnvoll, wenn ein Objekt unabhängig von einer absoluten Position innerhalb des Bildschirms bzw. des momentanen Fensters gezeichnet werden soll.

Grenzen — vor dem Aufruf von *MoveRel* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *LineRel, LineTo, MoveTo, GetX, GetY.*

Beispiel

```
begin
  ....
  MoveTo(1,2);
  MoveRel(10,10); { setzt den GC auf den Punkt (11,12) }
  ....
end.
```

MoveTo-Prozedur — Graph

Funktion — setzt den Grafik-Cursor auf einen bestimmten Punkt.

Deklaration
```
MoveTo( x, y: Integer)
```

Erläuterung — *MoveTo* setzt den Grafik-Cursor auf den durch die Integer-Ausdrücke *x* und *y* angegebenen Punkt. Wie alle anderen Grafik-Prozeduren arbeitet auch *MoveTo* relativ zu einem gesetzten Zeichenfenster. So setzt z.B. die nachstehende Befehlsfolge den Grafik-Cursor auf die (absoluten) Koordinaten *(120,60)*.

```
SetViewPort(100,40,400,140);
MoveTo(20,20);
```

Hinweis: Auch bei aktivierter Clip-Funktion kann der Grafik-Cursor auf beliebige Punkte gesetzt werden – das »Clipping« beeinflußt nur tatsächlich ausgeführte Zeichenoperationen, nicht die Positionierung des Grafik-Cursors selbst.

Grenzen — vor dem Aufruf von *MoveTo* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *GetMaxX, GetMaxY, GetX, GetY, MoveRel.*

Beispiel
```
begin
  ....
  MoveTo(0,0);  { setzt den GC in die obere linke Ecke
                  des Bildschirms bzw. Zeichenfensters }
  ....
end.
```

MsDos-Prozedur — Dos

Funktion — führt einen Funktionsaufruf von DOS aus.

Deklaration
```
MsDos(var RegsRegisters)
```

Erläuterung — der Aufruf der Prozedur *MsDos* hat dieselbe Wirkung wie ein Aufruf der Prozedur *Intr* mit *IntNo =$21*. Vor der Ausführung des Interrupts werden die Register des Prozessors mit den als *Regs* übergebenen Werten gesetzt, nach Ausführung des Interrupts schreibt *MsDos* die Register-Inhalte in die Felder dieses **var**-Parameters zurück. Der Datentyp *Registers* ist in der Unit *Dos* folgendermaßen definiert:

```
type
  Registers = record
    case Integer of
      0 : (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Word);
      1 : (AL,AZH,BL,BH,CL,CH,DL,DH: Byte);
    end;
```

Weitere Details finden Sie in Kapitel 14 »Interne Details« des *Programmierhandbuches*.

Grenzen Software-Interrupts, die bestimmte Werte der Register SS:SP voraussetzen oder diese Register verändern, können über *MsDos* nicht ausgeführt werden.

Querverweis *Intr*

New-Prozedur/Funktion

Funktion erzeugt eine dynamische Variable und setzt einen Zeiger darauf.

Deklaration
```
New(var p:Pointer)
New(var p: Pointer, Konstrukt)
```

Erläuterung In der ersten Form der Deklaration ist *p* eine Zeigervariable beliebigen Typs. *New* belegt einen Speicherbereich auf dem Heap, dessen Größe durch die Typen-Deklaration von *p* vorgegeben ist, und weist *p* die Startadresse dieses Bereichs zu. Wenn *p* beispielsweise als auf ein *Byte*-Array mit 6 Elementen zeigend deklariert ist (*var p : ^array[1..6] of Byte;*), dann belegt *New* automatisch 6 Bytes Platz auf dem Heap.

Speziell für objekt-orientierte Programmierung wurde die Syntax von *New* erweitert. In der zweiten Form der Deklaration ist *p* eine Zeigervariable auf einen Objekttyp und *Konstrukt* ein Aufruf der Konstruktor-Methode dieses Objekttyps. Diese erweiterte Syntax von *New* ist nichts anderes als eine Zusammenfassung der beiden Anweisungen:

```
New(P);
P^.Konstrukt;
```

Die erweiterte Syntax verbessert die Lesbarkeit und erzeugt außerdem einen kürzeren und effektiveren Code.

Eine weitere Erweiterung erlaubt die Verwendung von *New* als *Funktion*, die eine dynamische Variable eines bestimmten Typs – darin eingeschlossen Objekttypen – anlegt und einen Zeiger darauf zurückgibt. Die Syntax lautet dann: *New*(*P*) oder *New*(*P, Konstrukt*).

In der ersten Form kann *P* ein beliebiger Zeigertyp sein. In der zweiten Form muß *P* auf einen Objekttyp zeigen. *Konstrukt* ist der Aufruf des Konstruktors dieses Objekttyps. In beiden Fällen ist der Typ des Funktionsergebnisses *P*.

Wenn der Heap keinen Speicherbereich der gewünschten Größe zur Verfügung stellen kann, dann wird das Programm mit einem Laufzeitfehler abgebrochen. (Kapitel 14 »Interne Details« des *Programmierhandbuches* enthält weitere Details

und eine Beschreibung, wie sich derartige Fehler abfangen lassen).

Grenzen *New* und *Dispose* können nur unter bestimmten Umständen zusammen mit *Mark* und *Release* verwendet werden (siehe hierzu Kapitel 14 »Interne Details« des *Programmierhandbuches*).

Querverweis *GetMem*, *Dispose*, *FreeMem*, *Release*

NormVideo-Prozedur — Crt

Funktion setzt das Textattribut für nachfolgende Ausgaben, das beim Start des Programms gesetzt war.

Deklaration `NormVideo`

Erläuterung *NormVideo* setzt die (in der Unit *Crt* deklarierte) globale Variable *TextAttr* auf den Wert zurück, den das Zeichen auf der Position des Cursors beim Start des Programms hatte, d.h. die Zeichen- und Hintergrundfarben, die zu Anfang des Programms aktiv waren.

Querverweis *HighVideo*, *LowVideo*, *TextColor*, *TextBackground*.

NoSound-Prozedur — Crt

Funktion schaltet den eingebauten Lautsprecher ab.

Deklaration `NoSound`

Erläuterung *NoSound* beendet eine mit *Sound* gestartete Tonerzeugung.

Querverweis *Sound*.

Beispiel
```
begin
{ erzeugt einen Ton mit 440 Hz für eine halbe Sekunde: }
  Sound(440);  { Lautsprecher an }
  Delay(500);  { hier wären beliebige Befehle möglich }
  NoSound;     { Lautsprecher wieder aus }
end.
```

Odd-Funktion

Funktion prüft, ob das übergebene Argument eine ungerade Zahl darstellt.

Deklaration `Odd(x:LongInt)`

Ergebnistyp *Boolean*

Erläuterung *x* ist ein Ausdruck des Typs *LongInt* (und kann wegen der

automatischen Typ-Umwandlung ein beliebiger Teilbereich von *LongInt* wie *Word*, *ShortInt*, *Integer* usw. sein). *Odd* liefert den Wert *True* zurück, wenn *x* ungeradzahlig ist; ansonsten ist das Funktionsergebnis *False*.

Ofs-Funktion

Funktion liefert den Offset-Anteil einer Adresse zurück.

Deklaration `Ofs(x)`

Ergebnistyp *Word*

Erläuterung *x* steht für einen Variablenbezug beliebiger Art oder den Bezeichner einer Prozedur oder Funktion. Das Ergebnis von *Ofs* ist der *Offset*-Anteil der durch *x* angegebenen Adresse.

Querverweis *Seg, Addr*

Beispiel
```
var
  x : Integer;
begin
  Writeln('Die Variable x ist ab der Adresse ',
          Ofs(x), 'im Daten-Segment gespeichert.');
end.
```

Ord-Funktion

Funktion liefert die Ordinalzahl des Arguments zurück.

Deklaration `Ord(x)`

Ergebnistyp *LongInt*

Erläuterung *x* ist ein Ausdruck ordinalen Typs. Das Ergebnis der Funktion ist die Ordinalzahl von *x*.

Querverweis *Chr*

Beispiel
```
var
  Wurst: (Kohl, Birne);
begin
  Writeln(Ord(Kohl));       { 0 }
  Writeln(Ord('A'));      { 65 }
end.
```

OutText-Prozedur **Graph**

Funktion gibt Text an der Position des Grafik-Cursors aus.

Deklaration `OutText(Textstring : string)`

Erläuterung

Textstring ist ein String-Ausdruck und wird ab der momentanen Position des Grafik-Cursors ausgegeben. Ein automatischer Zeilenumbruch wie im Textmodus findet nicht statt: Wenn eine Ausgabe über den rechten Rand des Bildschirms bzw. des momentan gesetzten Zeichenfensters hinausgehen würde, dann hängt die Reaktion der Routine davon ab, welcher Zeichensatz momentan aktiv ist:

- Ausgaben mit Vektor-Zeichensätzen werden am rechten Rand des Bildschirms bzw. Zeichenfensters abgeschnitten;
- Ausgaben mit dem Standard-Zeichensatz werden *komplett* unterdrückt, wenn sie nicht vollständig auf den Bildschirm bzw. in das Zeichenfenster passen.

OutText benutzt die Schriftart und Formatierung, die durch *SetTextStyle* (Schriftart, Rotation, Größe) und *SetTextJustify* (linksbündig, zentriert, rechtsbündig) vorgegeben ist. Damit das Programm bei einem Wechsel der Schriftart und -größe nicht modifiziert werden muß, sollte der Platzbedarf auszugebender Strings immer über *TextWidth* und *TextHeight* berechnet werden.

Solange nicht mit Rotation gearbeitet wird, verändert *OutText* die X- und Y-Komponenten (GCX und GCY) abhängig von der Justierung in folgender Weise:

- Alle Kombinationen mit *LeftText* für die horizontale Formatierung:

```
GCX := GCX + TextWidth(TextString);
GCY := GCY;  { unverändert }
```

- Alle Kombinationen mit *RightText* oder *CenterText* für die horizontale Formatierung:

```
GCX := GCX;  { unverändert }
GCY := GCY + TextHeight(TextString);
```

Grenzen

vor dem Aufruf von *OutText* muß ein Grafikmodus gesetzt worden sein. Numerische Werte lassen sich mit *OutText* nur über eine vorherige Umwandlung in einen String (*Str*-Prozedur) ausgeben.

Querverweis

OutTextXY, *SetTextStyle*, *SetTextJustify*, *GetTextSettings*,*TextHeight*, *TextWidth*, *SetUserCharSize*.

Beispiel

```
program OutTextDemo;
uses Graph;
var
  GraphMode, GraphDriver: Integer;
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
```

```
  OutText('Ausgesprochen einfach!');
  Readln;
  CloseGraph;
end.
```

OutTextXY-Prozedur — Graph

Funktion — gibt Text an beliebigen Stellen des Grafik-Bildschirms aus.

Deklaration

```
OutTextxy( X, Y: Integer; Textstring : string)
```

Erläuterung — *Textstring* ist ein String-Ausdruck und wird ab dem durch die Integer-Ausdrücke (*X,Y*) festgelegten Punkt ausgegeben.

Ein automatischer Zeilenumbruch wie im Textmodus findet nicht statt: Wenn eine Ausgabe über den rechten Rand des Bildschirms bzw. des momentan gesetzten Zeichenfensters hinausgehen würde, dann hängt die Reaktion der Routine davon ab, welcher Zeichensatz momentan aktiv ist:

- Ausgaben mit Vektor-Zeichensätzen werden am rechten Rand des Bildschirms bzw. Zeichenfensters abgeschnitten;
- Ausgaben mit dem Standard-Zeichensatz werden *komplett* unterdrückt, wenn sie nicht vollständig auf den Bildschirm bzw. in das Zeichenfenster passen.

OutTextXY benutzt die Schriftart und Formatierung, die durch *SetTextStyle* (Schriftart, Größe) und *SetTextJustify* (linksbündig, zentriert, Rotation usw.) vorgegeben ist. Damit das Programm bei einem Wechsel der Schriftart und -größe nicht modifiziert werden muß, sollte der Platzbedarf auszugebender Strings immer über *TextWidth* und *TextHeight* berechnet werden.

Vor der Textausgabe wird der Grafik-Cursor auf die durch (*X,Y*) angegebene Position gesetzt. Durch die Ausgabe wird diese Position abhängig von der Justierung in folgender Weise verändert (Rotation ist hier nicht berücksichtigt):

- Alle Kombinationen mit *LeftText* für die horizontale Formatierung:

```
GCX := GCX + TextWidth(TextString);
GCY := GCY;  { unverändert }
```

- Alle Kombinationen mit *RightText* oder *CenterText* für die horizontale Formatierung:

```
GCX := GCX;  { unverändert }
GCY := GCY + TextHeight(TextString);
```

Grenzen — vor dem Aufruf muß ein Grafikmodus gesetzt worden sein.

Numerische Werte lassen sich mit *OutTextXY* nur über vorherige Umwandlung in einen String (*Str*-Prozedur) ausgeben.

Querverweis *OutText*, *SetTextStyle*, *SetTextJustify*, *GetTextSettings*, *TextHeight*, *TextWidth*, *SetUserCharSize*.

Beispiel

```
program OutTextXYDemo;
uses Graph;
var  GraphMode, GraphDriver: Integer;
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
  MoveTo(0,0);
  OutText('Etwas arg umständlich');
  OutTextXY(GetX,GetY, 'Auch nicht besser');
  Readln;
  OutTextXY(0,0, 'Perfekt !');  { überschreibt die }
  Readln;                       { erste Ausgabe }
  CloseGraph;
end.
```

OvrClearBuf-Prozedur — Overlay

Funktion löscht den Overlay-Puffer.

Deklaration `OvrClearBuf`

Erläuterung *OvrClearBuf* entfernt alle momentan geladenen Overlay-Units aus dem Hauptspeicher und erzwingt so ein Nachladen der entsprechenden Codesegmente aus der .OVR-Datei bzw. der EMS-Karte für folgende Overlay-Aufrufe.

Diese Routine muß im »Normalbetrieb« nie aufgerufen werden – sie ist einzig und allein für sehr spezielle Anwendungen gedacht, bei denen der Puffer explizit neu geordnet oder kurzfristig für andere Zwecke freigegeben werden soll.

Hinweis: Wenn *OvrClearBuf* von einem Overlay aus aufgerufen wird, entfernt die Overlay-Verwaltung auch dieses Overlay aus dem Speicher – und lädt es sofort nach dem Rücksprung von *OvrClearBuf* erneut.

Querverweis *OvrGetBuf*, *OvrSetBuf*

OvrGetBuf-Funktion — Overlay

Funktion liefert die momentane Größe des Overlay-Puffers zurück.

Deklaration `OvrGetBuf`

Ergebnistyp *LongInt*

Erläuterung *OvrGetBuf* liefert die momentane Größe des Overlay-Puffers in

Bytes. Standardmäßig ist dieser Puffer so bemessen, daß die größte vom Programm verwendete Overlay-Unit genau hineinpaßt.

Hinweis: Obwohl diese Standardgröße auf eine einzelne Unit zugeschnitten ist, kann sie 64 KByte übersteigen, weil hier nicht nur der Code des Overlays, sondern auch Tabellen mit Querverweisen gespeichert werden müssen.

Der Overlay-Puffer wird durch den Initialisierungsteil der Unit *Overlay* belegt; seine Größe läßt sich über die Prozedur *OvrSetBuf* direkt verändern.

Querverweis *OvrInit*, *OvrInitEMS*, *OvrSetBuf*

OvrGetRetry-Funktion — Overlay

Funktion ermittelt die aktuelle Größe des Bewährungsbereichs.

Deklaration `OvrGetRetry`

Ergebnistyp *Longint*

Erläuterung Die Funktion gibt den Wert zurück, der in *OvrSetRetry* zuletzt gesetzt wurde.

Querverweis *OvrSetRetry*

OvrInit-Prozedur — Overlay

Funktion initialisiert die Overlay-Verwaltung und öffnet die .OVR-Datei des Programms.

Deklaration `OvrInit(Filename : string)`

Erläuterung der String-Ausdruck *Filename* muß die Datei angeben, in der die Overlay-Units des Programms gespeichert sind. (Der Dateiname ergibt sich normalerweise aus dem Namen der .EXE-Datei und dem Suffix .OVR).

Wenn *Filename* keinen Suchweg und/oder Laufwerksbezeichner enthält, dann sucht *OvrInit* zuerst im momentan gesetzten Verzeichnis, dann in dem Verzeichnis, in dem die .EXE-Datei gespeichert ist (Voraussetzung: DOS-Version 3.x) und schließlich in sämtlichen Verzeichnissen, die über PATH angegeben sind.

Nach der Bestimmung, wieviel Platz die größte Overlay-Unit beansprucht, wird der Heap-Beginn entsprechend in Richtung höherer Speicheradressen verschoben und so Platz für den Overlay-Puffer geschaffen.

Im Fehlerfall wird die Overlay-Verwaltung nicht installiert – nachfolgende Aufrufe von Overlay-Routinen haben dann

den Abbruch des Programms mit dem Fehlercode 208 zur Folge.

Achtung: Alle anderen Routinen von *Overlay* (inklusive *OvrInitEMS*) setzen voraus, daß die Overlay-Verwaltung durch einen Aufruf von *OvrInit* initialisiert wurde.

Querverweis *OvrGetBuf*, *OvrInitEMS*, *OvrSetBuf*

Beispiel

```
program OvrInitDemo;
uses Overlay;
begin
  OvrInit('OVRDEMO.OVR');
  if OvrResult <> ovrOk then
  begin
    case OvrResult of
      ovrError: Writeln('Programm hat keine Overlays.');
      ovrNotFound: Writeln('OVR-Datei nicht gefunden.');
    end;
    Halt(1);
  end
   else Writeln('Overlay-Verwaltung initialisiert.');
end.
```

OvrInitEMS-Prozedur Overlay

Funktion kopiert die Overlay-Datei des Programms in eine EMS-Karte.

Deklaration `OvrInitEMS`

Erläuterung *OvrInitEMS* prüft, ob das System mit einer EMS-Karte ausgerüstet ist, die genug Platz für die Overlay-Datei bietet: Wenn ja, wird die .OVR-Datei vollständig in die EMS-Karte geladen, weitere Diskettenzugriffe durch die Overlay-Verwaltung sind dann nicht mehr notwendig.

Weil sich Programmteile nicht *in* der EMS-Karte ausführen lassen, bleibt das restliche Konzept (Overlay-Puffer, abwechselndes Laden von Overlay-Units usw.) dasselbe – *OvrInitEMS* ersetzt »lediglich« Diskettenzugriffe durch Kopieraktionen innerhalb des Speichers und beschleunigt so den Wechsel zwischen Overlays um ein Vielfaches.

Hinweise: Ein erfolgloser Aufruf von *OvrInitEMS* hat keine negativen Seiteneffekte für das Programm – die Overlay-Verwaltung arbeitet in diesem Fall auf normale Weise weiter (d.h. mit Diskettenzugriffen).

Wenn Sie Ihre EMS-Karte als RAM-Disk verwenden, sollten Sie (über eine entsprechende Änderung der Initialisierung des EMS-Treibers in der Datei CONFIG.SYS) sicherstellen, daß auf der Karte genügend *unbelegter* Platz für die Overlay-Verwaltung von Turbo Pascal zur Verfügung steht.

Grenzen *OvrInitEMS* setzt minimal die Version 3.2 des EMS-Treibers voraus.

Querverweis *OvrGetBuf*, *OvrInit*, *OvrSetBuf*

OvrSetBuf-Prozedur — Overlay

Funktion legt die Größe des Overlay-Puffers explizit fest.

Deklaration
```
OvrSetBuf(Size: LongInt)
```

Erläuterung *OvrSetBuf* legt die Größe des Overlay-Puffers auf *Size* Bytes fest: Je größer der Puffer ist, desto mehr Overlay-Units lassen sich gleichzeitig im Speicher halten.

Der Heap-Beginn wird durch *OvrSetBuf* entsprechend der neuen Größe des Puffers nach unten bzw. oben verschoben. Zum Zeitpunkt des Aufrufs von *OvrSetBuf* dürfen deshalb keine dynamischen Variablen belegt sein – falls doch, endet *OvrSetBuf* mit dem Statuscode *ovrError* und läßt den Puffer unverändert. (*New* und *GetMem* setzen ein spezielles Flag für die Overlay-Verwaltung – deshalb können Fehler dieser Art zuverlässig erkannt werden).

Die Minimalgröße des Puffers ist durch das umfangreichste Overlay des Programms vorgegeben, die Maximalgröße durch den momentanen Pufferumfang plus dem noch freien Platz auf dem Heap (der sich in diesem Fall durch *MemAvail* oder *MaxAvail* ermitteln läßt). Wenn die durch *Size* angegebene Größe außerhalb dieses Bereichs liegt, setzt *OvrSetBuf* den Status *ovrError* (Puffer läßt sich nicht auf diese Größe verkleinern) bzw. *ovrNoMemory*. Negative Seiteneffekte gibt es auch hier nicht – die Overlay-Verwaltung arbeitet einfach mit der alten Puffergröße weiter.

Querverweis *OvrGetBuf*

OvrSetRetry-Prozedur — Overlay

Funktion legt die Größe des »Bewährungsbereichs« im Overlay-Puffer fest.

Deklaration
```
OvrSetRetry(Groesse : Longint)
```

Erläuterung Die Prozedur *OvrSetRetry* legt die Größe des »Bewährungsbereichs« im Overlay-Puffer fest, wobei der Parameter *Groesse* angibt, wieviel Bytes *vor* dem Ende des Puffers der Bewährungsbereich beginnt. Wird nun ein Overlay in den so definierten Bereich verschoben, beginnt für dieses Overlay die Bewährungsphase. Aus Gründen der Kompatibilität zu früheren Versionen der Overlay-Verwaltung ist die Standardgröße des

Bewährungsbereichs 0, wodurch der Bewährung/Aufschub-Mechanismus im Endeffekt ausgeschaltet wird.

Es gibt (noch) keine empirische Formel, mit der man die optimale Größe des Bewährungsbereichs ermitteln könnte. Versuche haben jedoch gezeigt, daß Werte, die von einem Drittel bis zur Hälfte des Overlay-Puffers reichen, die besten Ergebnisse liefern.

Querverweis *OvrGetRetry*

Beispiel
```
OvrInit('MEINPROG.OVR');
OvrSetBuf(Puffergroesse);
OvrSetRetry(Puffergroesse div 3);
```

PackTime-Prozedur — Dos

Funktion konvertiert einen Record des Typs *DateTime* in einen »gepackten« *LongInt* für die Verwendung in *SetFTime*.

Deklaration
```
PackTime(var DT: DateTime; var  Time: LongInt)
```

Erläuterung der Datentyp *DateTime* ist in der Unit *Dos* folgendermaßen deklariert (vgl. Kapitel 12 »Standard Units« des *Programmierhandbuches*):

```
type
  DateTime = record
      Year, Month,   { Jahr 1980..2099, Monat 1..12 }
      Day, Hour,     { Tag 1..31, Stunde 0..23 }
      Min, Sec: Word;  { Minuten, Sekunden 0..59 }
    end;
```

Grenzen *PackTime* prüft nicht, ob die »Felder« der übergebenen *LongInt*-Variablen zulässige Werte für eine Konvertierung enthalten.

Querverweis *UnpackTime*, *GetFTime*, *GetTime*, *SetFTime*, *SetTime*

ParamCount-Funktion

Funktion liefert die Anzahl der Kommandozeilen-Parameter zurück, die dem Programm übergeben wurden.

Deklaration
```
ParamCount
```

Ergebnistyp *Word*

Erläuterung Leerzeichen und Tabs werden als Trennung zwischen zwei Parametern betrachtet, der Name des Programms selbst wird von *ParamCount* nicht mitgezählt.

Hinweis: Ein »Einklammern« von Parametern (mit Anführungszeichen oder Hochkommas) hat keine Wirkung. Beim Aufruf eines Programms wie

```
Paramtst eins "zwei drei" vier
```

liefert *ParamCount* das Ergebnis 4 zurück – die Zeichenfolgen `"zwei` und `drei"` werden als zweiter und dritter Parameter betrachtet.

Querverweis *ParamStr*

Beispiel

```
var
  x : Word;

begin
  if ParamCount = 0 then
    Writeln('Heute keine Parameter.')
  else
   for x:= 1 to ParamCount do
    Writeln('Parameter ',x,': ', ParamStr(x));
end.
```

ParamStr-Funktion

Funktion liefert einen Kommandozeilen-Parameter zurück.

Deklaration `ParamStr(index: Word)`

Ergebnistyp *String*

Erläuterung *index* ist ein Ausdruck des Typs *Word* und gibt die Nummer des Kommandozeilen-Parameters an, der von *ParamStr* zurückgeliefert werden soll.

Wenn *index* größer als *ParamCount* ist, dann liefert *ParamStr* einen Nullstring als Ergebnis.

Hinweis: Beim Betrieb unter DOS 3.x liefert *ParamStr(0)* Suchweg und Namen des laufenden Programms. Unter DOS 2.x hat *ParamStr(0)* einen Nullstring als Ergebnis.

Querverweis *ParamCount, FSplit*

Beispiel

```
program ProgNameDemo;  { Datei PNDEMO.PAS }
uses DOS;

begin
  if Lo(DosVersion) < 3 then
     Writeln('DOS-Version 3.x wird vorausgesetzt!')
    else Writeln('Programmname: ', ParamStr(0));
                 { z.B. 'D:\TP50\PROGRAMS\PNDEMO.EXE' }
end.
```

Pi-Funktion

Funktion liefert den Wert der mathematischen Konstanten *Pi* zurück (3.141592653589793285).

Deklaration `Pi`

Ergebnistyp *Real*

Erläuterung das Ergebnis der Funktion ist der Wert von *Pi*.

Hinweis: Die Genauigkeit des oben angegebenen Wertes wird nur bei Verwendung eines numerischen Coprozessors bzw. des Emulators erreicht – im Modus {*SN-*} liefert *Pi* den Wert 3.1415926536, rundet also ab der neunten Stelle nach dem Dezimalpunkt.

Querverweis *ArcTan*, *Cos*, *Sin*

PieSlice-Prozedur Graph

Funktion zeichnet ein ausgefülltes »Kuchenstück«.

Deklaration `PieSlice(X, Y: Integer; StAngle , EndAngle , Radius: Word)`

Erläuterung *PieSlice* zeichnet ein Kreisbogenstück, dessen Mittelpunkt durch die Integer-Ausdrücke (*X,Y*) angegeben ist. *StAngle* legt den Startpunkt des mit *Radius* gezogenen Kreisbogens fest, *EndAngle* sein Ende. Die Angabe beider Winkel erfolgt in *Grad*, wobei die Zählung entgegen dem Uhrzeigersinn läuft (vgl. *Arc*-Prozedur).

Die Endpunkte des Kreisbogens werden mit dem Kreismittelpunkt verbunden, danach füllt *PieSlice* das entstandene »Kuchenstück« mit dem durch *SetFillPattern* bzw. *SetFillStyle* festgelegten Muster (und in der dort angegebenen Farbe). Wenn während des Ausfüllens ein Fehler auftritt, liefert der nächste Aufruf von *GraphResult* den Wert -6.

Der Aufruf von *PieSlice* mit den Werten 0 und 360 für *StAngle* und *EndAngle* zeichnet einen vollständigen, ausgefüllten Kreis.

Hinweis: Jeder Grafiktreiber (.BGI) enthält einen Wert zur Korrektur der physikalischen Höhen-/Seitenverhältnisse des Bildschirms (*AspectRatio*), der automatisch von *Arc*, *Circle* und *PieSlice* verwendet wird, um auch optisch runde Kreise zu erhalten. Mit *GetAspectRatio* läßt sich dieser Korrekturfaktor ermitteln, mit *SetAspectRatio* direkt setzen.

Grenzen vor dem Aufruf von *PieSlice* muß ein Grafikmodus gesetzt worden sein.

Querverweis *Arc*, *Circle*, *Ellipse*, *GetArcCoords*, *GetAspectRatio*, *SetAspectRatio*, *SetFillStyle*, *SetFillPattern*.

```
program PieSliceDemo;
uses Graph;
var
  GraphMode, GraphDriver: Integer;
  x : Integer;

begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');

                      { ein Seeschneckenhaus }
  for x:= 0 to 35 do
    PieSlice(80,60, x * 10,
             round((x + 0.5) * 10), (x + 1) * 3);
  Readln;
  CloseGraph;
end.
```

Pos-Funktion

Funktion	sucht einen String nach einer Zeichenfolge ab.
Deklaration	`Pos(substr : string ; s: string)`
Ergebnistyp	*Byte*
Erläuterung	*Pos* sucht den als *s* übergebenen String-Ausdruck nach dem ersten Vorkommen des String-Ausdrucks *substr* ab. Wenn *substr* in *s* vollständig enthalten ist, dann liefert *Pos* die Position des Zeichens in *s* zurück, mit dem *substr* beginnt. Wenn *substr* nicht in *s* enthalten ist, dann hat *Pos* das Funktionsergebnis 0.
Querverweis	*Copy, Delete, Insert, Length*
Beispiel	

```
var
  s : string;
begin
  s := 'ABCDEF';
  Writeln(Pos('BCD', s));   { Ausgabe: 2 }
  Writeln(Pos('ACD', s));   { Ausgabe: 0 }
  Writeln(Pos('EFG', s));   { Ausgabe: 0 }
end.
```

Pred-Funktion

Funktion	liefert den Vorgänger des Arguments zurück.
Deklaration	`Pred( x)`

Ergebnistyp derselbe Typ wie das Argument.

Erläuterung *x* ist ein Ausdruck ordinalen Typs. Das Funktionsergebnis ist der Vorgänger von *x* (und hat denselben Typ).

Grenzen wenn *x* das Element mit dem niedrigsten ordinalen Wert innerhalb seines Typs darstellt, dann ist der Vorgänger von *x* nicht definiert und *Pred* erzeugt einen Laufzeitfehler.

Querverweis *Dec*, *Inc*, *Succ*

Ptr-Funktion

Funktion konvertiert zwei Angaben für *Segment* und *Offset* in einen Wert des Typs *Pointer*.

Deklaration
```
Ptr(Seg, Ofs: Word)
```

Ergebnistyp *Pointer*

Erläuterung *Seg* und *Ofs* sind Ausdrücke des Typs *Word*. Das Funktionsergebnis ist ein Zeiger, der auf die durch *Seg* und *Ofs* angegebene Adresse zeigt (*Segment:Offset*). Wie die Konstante **nil** ist das Ergebnis von *Ptr* ein »generischer Zeiger« und mit allen anderen Zeiger-Typen kompatibel.

Das Ergebnis von *Ptr* kann zusammen mit dem Operator ^ für den direkten Zugriff auf Speicherzellen des Computers verwendet werden:

```
Writeln('Der momentan gesetzte Videomodus ist: ',
  Byte(Ptr($40,$49)^ ) );
```

Wie in Kapitel 3 »Typen« des *Programmierhandbuches* erläutert, erfordern generische Zeigerwerte eine explizite Typ-Umwandlung – ansonsten kann der Compiler nicht bestimmen, ob von der angegebenen Adresse ein *Byte*, *Integer*, *Word* usw. gelesen werden soll. Bei Zuweisungen des Ergebnisses von *Ptr* an *bereits definierte* Zeiger-Variablen »weiß« der Compiler dagegen, wie der Wert behandelt werden muß, und nimmt eine automatische (implizite) Konvertierung vor – siehe »Beispiel«.

Querverweis *Addr*

Beispiel
```
var
  p : ^Byte;  { Zeiger auf Variablen des Typs Byte }
begin
  p := Ptr($40,$49);  { implizit konvertiert }
  Writeln('Momentaner Videomodus: ', p^);
end.
```

PutImage-Prozedur Graph

Funktion kopiert den Inhalt einer Puffervariablen bitweise in einen rechteckigen Bildausschnitt, wobei logische Verknüpfungen mit dem Inhalt dieses Ausschnitts möglich sind.

Deklaration
```
PutImage( x, y: Integer; var  BitMap ; BitBlt: Word)
```

Erläuterung *(x,y)* sind Ausdrücke des Typs *Integer* und legen die obere linke Ecke des zu kopierenden Bildausschnitts fest. *BitMap* ist ein untypisierter Parameter, den *PutImage* als Startadresse des Speicherbereichs interpretiert, aus dem die Bilddaten kopiert werden sollen.

Die ersten beiden 16-Bit-Worte dieses Speicherbereichs müssen die Breite und Höhe des zu kopierenden Ausschnitts enthalten:

```
Breite := x2 - x1 + 1;
Hoehe:= y2 - y1 + 1;
```

Insgesamt enthält ein mit *GetImage* gespeicherter Bildausschnitt also vier Bytes mehr als direkt für die Bilddaten notwendig sind (was die Funktion *ImageSize* automatisch berücksichtigt).

Der Parameter *BitBlt* (Bit-Block Transfer) ist ein Ausdruck des Typs *Word* und legt fest, welche Verknüpfungen *PutImage* mit dem Inhalt der Puffervariablen und dem momentanen Inhalt des Ziel-Bildausschnitts ausführt.

Die Unit *Graph* definiert für *BitBlt* die folgenden Konstanten:

```
const
  CopyPut    = 0;   { Überschreiben (MOV-Befehl) }
  NormalPut  = 0;   { wg. Kompatibilität mit V 4.0 }
  XorPut     = 1;   { XOR-Operation mit Bildinhalt }
  OrPut      = 2;   { OR-Operation }
  AndPut     = 3;   { AND-Operation }
  NotPut     = 4;   { NOT-Operation, d.h. Überschrei-
                      ben mit invertierten Werten }
```

Ein Aufruf der Form *PutImage(x,y,BitMap,CopyPut)* ignoriert also den Inhalt des Bildschirms und überschreibt die dort gespeicherten Daten. Mit dem Modus *XorPut* lassen sich bewegte Grafiken recht einfach erzeugen – eine erste Kopie läßt den Inhalt der Puffervariablen auf dem Bildschirm erscheinen, die zweite Kopie läßt ihn wieder verschwinden und bringt den ursprünglichen Bildausschnitt wieder zum Vorschein.

Grenzen vor dem Aufruf von *PutImage* muß ein Grafikmodus gesetzt worden sein. Die durch *(x,y)* und *x+Breite* festgelegten Koordi-

naten müssen sich innerhalb des Bildschirms bzw. des momentan gesetzten Zeichenfensters befinden, sonst bricht *PutImage* ab, ohne eine Operation auszuführen. (Ein automatisches »Clipping« findet nur in vertikaler Richtung statt, d.h. für Zeilen, die über den unteren Rand des Bildschirms bzw. Zeichenfensters hinausgehen würden).

Querverweis *GetImage, ImageSize.*

Beispiel

```
program PutImageDemo;
uses Graph;
var
  GraphMode, GraphDriver: Integer;
  p : Pointer; Size, MaxX: Word;
  x : Integer;
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
  MaxX := GetMaxX div 4;      { ein Viertel des
                                Bildschirms }
  for x := 1 to 100 do begin
    SetColor(Random(16));     { ein paar bunte Linien }
    LineTo(Random(MaxX), Random(GetMaxY));
   end;
  Readln;
  Size := ImageSize(0,0,MaxX,GetMaxY); { Größe
                                         bestimmen }
  GetMem(p, Size);            { Platz auf dem Heap }
  GetImage(0,0,MaxX,GetMaxY,p^);
  for x:= 1 to 3 do
    PutImage(x * MaxX, 0, p^, NormalPut); { 3 Kopien }
  Readln;
  CloseGraph;
end.
```

PutPixel-Prozedur — Graph

Funktion zeichnet einen Punkt auf den angegebenen Koordinaten.

Deklaration `PutPixel( x, y: Integer; Color: Word)`

Erläuterung *x* und *y* sind Integer-Ausdrücke und legen die Stelle des Bildschirms fest, auf der das Pixel mit der durch *Color* festgelegten Farbe gezeichnet werden soll.

Grenzen vor dem Aufruf von PutPixel muß ein Grafikmodus gesetzt worden sein.

Querverweis *GetImage, GetPixel, PutImage.*

Beispiel

```
begin
```

```
  ....
  Randomize;
  repeat   { zeichnet einen bunten Sternenhimmel }
   PutPixel(Random(GetMaxX), Random(GetMaxY),
            Random(16));
  until KeyPressed;
  ....
end.
```

Random-Funktion

Funktion	liefert eine »Zufallszahl« zurück.
Deklaration	`Random [ (Range: Word) ]`
Ergebnistyp	*Real* oder *Word*, abhängig von der Art des Aufrufs.
Erläuterung	wenn *Range* nicht angegeben ist, liefert *Random* einen Realwert zurück, der die Bedingung $0 <= Ergebnis < 1$ erfüllt. Wenn *Random* dagegen einen Integer-Ausdruck als *Range* übergeben bekommt, dann liefert diese Funktion einen positiven Integer zurück, der die Bedingung $0 <= Ergebnis < Range$ erfüllt. Ein Aufruf von *Random* mit dem Wert 0 für *Range* hat immer das Ergebnis 0. Damit *Random* nicht bei jedem Programmlauf dieselbe Folge von »Zufallszahlen« liefert, sollte der Zufallszahlengenerator mit *Randomize* zu Beginn des Programms initialisiert werden.
Querverweis	*Randomize*
Beispiel	

```
uses Crt;
begin
  Randomize;  { Initialisierung des Generators }
  repeat
    TextAttr := Random(256);  { erzeugt einen sehr }
    Write('!');               { bunten Bildschirm }
  until KeyPressed;
end.
```

Randomize-Prozedur

Funktion	initialisiert den eingebauten »Zufallszahlen«-Generator.
Deklaration	`Randomize`
Erläuterung	*Randomize* verwendet Datum und Uhrzeit des Systems, um einen Startwert für folgende Aufrufe von *Random* zu setzen. Programme, die ohne *Randomize* arbeiten, erzeugen bei jedem Start immer wieder dieselbe Folge von »Zufallszahlen«.

Hinweis: Der Startwert des Generators ist in einer globalen *LongInt*-Variablen namens *RandSeed* gespeichert – durch direkte Zuweisungen an diese Variable läßt sich eine Folge von »Zufallszahlen« beliebig oft wiederholen (z.B. für Testzwecke und Verschlüsselungsprogramme).

Querverweis *Random*

Beispiel siehe *Random*

Read-Prozedur

Funktion liest eine oder mehrere Komponenten aus einer typisierten Datei bzw. einen oder mehrere Werte aus einer Textdatei in die angegebenen Variablen.

Deklaration

```
Read(var f,v1 [,v2, ...,vn])                    für typisierte Dateien
Read([var f: Text;] v1 [,v2,...,vn])            für Textdateien
```

Erläuterung *Read für typisierte Dateien*: *f* steht für eine Datei-Variable beliebigen Typs (außer *Text*), jede der angegebenen Variablen *v* hat denselben Typ wie die Komponenten dieser Datei. Der Positionszeiger innerhalb der Datei wird durch jede Leseaktion um eine Komponente in Richtung auf das Dateiende bewegt. Der Versuch, über das Ende der Datei »hinauszulesen« (d.h. der Aufruf von *Read*, wenn *Eof(f)* den Wert *True* ergibt), erzeugt einen Laufzeitfehler.

Wenn *Read* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus `{$I-}` den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode ($<>0$) als Ergebnis.

Read für Textdateien: wenn *f* angegeben ist, muß dieser Parameter für eine Textdatei-Variable stehen – ist *f* nicht angegeben, dann liest *Read* von der Standard-Eingabe (*Input*).

Für die als *v* angegebenen Variablen sind die Typen *Char*, *String* sowie sämtliche Integer- und Real-Typen zulässig, die Mischung mehrerer Typen in einem Aufruf von *Read* ist erlaubt.

Abhängig vom Typ der angegebenen Variablen interpretiert *Read* gelesene Daten auf unterschiedliche Art und Weise:

Char-Variablen: Read liest ein Zeichen aus der Datei und ordnet es der angegebenen Variablen zu. Wenn *Eof(f)* vor dem Aufruf von *Read* den Wert *True* ergibt (d.h. das Dateiende erreicht ist), dann bekommt die *Char*-Variable das Zeichen *chr(26)* (= Ctrl Z) zugewiesen. Wenn *Eoln(f)* vor dem Aufruf von *Read* den Wert *True* ergibt (d.h. ein Zeilenende erreicht ist), dann

bekommt die *Char*-Variable das Zeichen *chr(13)* (=Zeilenvorschub) zugewiesen.

Der Positionszeiger wird um ein Zeichen in Richtung auf das Dateiende versetzt – aufeinanderfolgende Aufrufe von *Read* mit *Char*-Variablen lesen also die Datei Zeichen für Zeichen.

Integer-Variablen: Read erwartet eine Folge numerischer Zeichen, die den in Kapitel 1 »Sprachelemente und Konstanten« des *Programmierhandbuches* angegebenen Regeln entspricht. Führende Leerzeichen, Tabs und Zeilenvorschübe werden automatisch übersprungen, das erste nicht-numerische Zeichen (oder das Erreichen des Dateiendes) bricht den Lesevorgang ab.

Wenn die Interpretation der gelesenen Zeichenfolge einen Wert des erwarteten Formats (*Byte*, *ShortInt*, *Integer* usw.) ergibt, wird dieser Wert der angegebenen Variablen zugewiesen, ansonsten erzeugt *Read* einen Laufzeitfehler.

Wenn das Dateiende vor dem Aufruf von *Read* bereits erreicht war oder *Eof(f)* während des Überspringens führender Leerzeichen, Tabs und Zeilenvorschübe den Wert *True* annimmt, weist *Read* der angegebenen Variablen den Wert 0 zu.

Der Positionszeiger bleibt nach der Leseaktion auf dem Zeichen stehen, durch das die Leseaktion abgebrochen wurde – der nächste Aufruf von *Read* liest also dieses Zeichen als erstes und überspringt es, wenn wiederum ein numerischer Wert erwartet wird.

Real-Variablen: Read erwartet eine Folge numerischer Zeichen, die den in Kapitel 1 »Sprachelemente und Konstanten« des *Programmierhandbuches* definierten Regeln entspricht. Im Unterschied zu Integerwerten stellen der Dezimalpunkt und das »E« für einen Exponenten hier gültige Zeichen dar, hexadezimale Ziffern dagegen nicht. Führende Leerzeichen, Tabs und Zeilenvorschübe werden automatisch übersprungen, das erste nicht-numerische Zeichen (oder das Erreichen des Dateiendes) bricht den Lesevorgang ab.

Wenn die Interpretation der gelesenen Zeichenfolge einen Wert des erwarteten Formats (*Byte*, *ShortInt*, *Integer* usw.) ergibt, wird dieser Wert der angegebenen Variablen zugewiesen, ansonsten erzeugt *Read* einen Laufzeitfehler.

War das Dateiende vor dem Aufruf von *Read* bereits erreicht oder nimmt *Eof(f)* während des Überspringens führender Leerzeichen, Tabs und Zeilenvorschübe den Wert *True* an, weist *Read* der angegebenen Variablen den Wert 0.0 zu.

Der Positionszeiger bleibt nach der Leseaktion auf dem Zeichen stehen, durch das die Leseaktion abgebrochen wurde – der nächste Aufruf von *Read* liest also dieses Zeichen als erstes und überspringt es, wenn wiederum ein numerischer Wert erwartet wird.

String-Variablen: Read liest solange Zeichen ab der momentanen Position, bis *Eoln(f)* den Wert *True* annimmt (d.h. ein Zeilenende erreicht ist). Sämtliche gelesenen Zeichen außer dem Zeilenende selbst werden der String-Variablen zugeordnet.

Wenn zwischen der momentanen Position und dem nächsten Zeilenende mehr als 255 Zeichen liegen, liest *Read* dennoch bis zum nächsten Zeilenende (wobei kein Laufzeitfehler erzeugt wird). Die String-Variable bekommt in diesem Fall die *ersten* 255 gelesenen Zeichen zugewiesen, die folgenden Zeichen bis zum Zeilenende werden ignoriert.

Der Positionszeiger innerhalb der Datei bleibt nach der Leseaktion auf dem Zeilenende stehen – der nächste Aufruf von *Read* liest also das Zeilenende als erstes Zeichen.

Hinweis: Da *Read* Zeilenenden *nicht* automatisch überspringt, brechen aufeinanderfolgende Leseaktionen mit String-Variablen ab dem zweiten Aufruf immer wieder am selben Zeilenende ab und liefern einen Nullstring zurück. Zum Lesen aufeinanderfolgender Textzeilen muß deshalb die Prozedur *Readln* verwendet werden.

Wenn *Read* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Grenzen die durch *f* angegebene Datei muß für Leseoperationen eröffnet worden sein.

Querverweis *Readln, ReadKey, Write, Writeln*

Beispiel

```
program ReadDemo;
type
  Datensatz = record
      Name : string[20];
      Alter: Byte;
    end;

var
  FTyp : file of Datensatz;  { typisierte Datei }
  ByteDatei: file of Byte;
  EinSatz: Datensatz;
  b: Byte;
  x,y: Integer;
  z: Real;
  s: string;
begin
{  .... }
  while not Eof(FTyp) do  { solange die Datei noch
                            Komponenten  enthält... }
```

```
  begin
    Read(FTyp, EinSatz); { liest jeweils Name und
                           Alter }
    Writeln('Name: ', EinSatz.Name,
            ' Alter: ', EinSatz.Alter);
  end;
{  ... }
  while not Eof(ByteDatei) do   { liest eine beliebige
                                  Datei Byte für Byte }
  begin
    Read(ByteDatei, b);   { und gibt ihren Inhalt als }
    Write(chr(b));        { ASCII-Zeichen aus }
  end;

  Readln(Input, s); { liest Zeichen von der Tastatur
                      bis zum nächsten [¿] }
  Writeln(s);
  Read(x); Read(y); Read(z);
  { dasselbe wie Read(Input, x,y,z); :  }
  { erwartet zwei Integer, eine Realzahl,
    z.B. 34 2643 4.349587 <RETURN>
    oder 2234 <RET> 5498 <RET> 567.45867 <RET> }
  Writeln(x:5,y:5,z:8);
end.
```

ReadKey-Funktion — Crt

Funktion — liest ein Zeichen von der Tastatur, ohne es automatisch auf den Bildschirm auszugeben.

Deklaration — `ReadKey`

Ergebnistyp — *Char*

Erläuterung — wenn *KeyPressed* vor dem Aufruf von *ReadKey* den Wert *True* liefert (d.h. bereits eine »noch nicht gelesene« Taste gedrückt wurde), dann liefert *ReadKey* das entsprechende Zeichen ohne Verzögerung zurück; ansonsten wird die Programmausführung solange angehalten, bis der Benutzer eine Eingabe macht.

Spezialtasten wie Funktions- und Cursortasten sowie Kombinationen mit Alt erzeugen *erweiterte Tastaturcodes*. Der erste Aufruf von *ReadKey* liefert nach derartigen Eingaben das Zeichen *chr(0)*, der zweite Aufruf liefert den *Scancode* der entsprechenden Taste. Das Zeichen *chr(0)* wird ausschließlich als Einleitung für Spezialtasten verwendet: wenn ein erster Aufruf von *ReadKey* dieses Ergebnis hat, kann das Programm mit Sicherheit davon ausgehen, daß (a) ein weiterer Tastaturcode anliegt und (b) dieser Code der Scancode einer Spezialtaste ist.

Hinweis: Wenn die globale Variable *CheckBreak* auf *False* gesetzt ist, liefert *ReadKey* das Zeichen *chr(3)* (= ^C) zurück, wenn der Benutzer die Tastenkombination Ctrl Break eingibt. Ist *CheckBreak* dagegen auf *True* gesetzt, dann bricht die Eingabe von Ctrl Break das Programm ab.

Querverweis *KeyPressed*

Beispiel

```
program ReadKeyDemo;
uses Crt;

var
  ch: Char;
begin
  ch := ReadKey;
  if ch = #0 then
  begin
    write('Funktionstaste. Scancode: ');
    writeln(ord(ReadKey));  { zweiter ReadKey-Aufruf }
  end
   else writeln('normale Taste: ', ch);
end.
```

Readln-Prozedur

Funktion führt einen Aufruf von *Read* aus und springt dann zum Anfang der nächsten Zeile innerhalb der angegebenen Datei.

Deklaration `Readln( [var f: Text;] v1 [,v2,...,vn])`

Erläuterung *Readln* ist eine Erweiterung der Prozedur *Read*, die sich nur auf Textdateien anwenden läßt.

Wenn *f* angegeben ist, muß es sich dabei um eine Textdatei-Variable handeln; ist *f* nicht angegeben, dann liest *Readln* die Standard-Eingabe (*Input*).

Aufrufe der Form *Readln* und *Readln(f)* – d.h. ohne Angaben von Ziel-Variablen – führen eine Leseaktion bis zum nächsten Zeilenende durch (oder dem Ende der Datei, falls kein Zeilenende gefunden wird) und ignorieren alle dabei gelesenen Zeichen.

Zur Prozedur *Read* besteht ein grundlegender Unterschied: *Readln* erwartet unabhängig von der Anzahl der angegebenen Variablen *einen* Zeilenvorschub am Ende der Leseaktion. Wenn wir annehmen, daß der Positionszeiger in einer Textdatei *f* auf den Anfang der Zeile

```
123 <Tab> 44 <Tab> 9999 <Zeilende>
```

zeigt, dann setzt ihn *jeder* der drei folgenden Aufrufe von *Readln* hinter das Ende dieser Zeile:

```
Readln(f, x1); {liest den 1.Wert, überspringt den Rest}
Readln(f, x2, x2); {liest 2 Werte, überspringt den 3.}
Readln(f, x1, x2, x3); {liest alle drei Werte}
```

Bei der Verwendung von *Read* anstelle von *Readln* würde die Position innerhalb der Datei dagegen jeweils hinter dem zuletzt gelesenen Wert stehenbleiben (also im ersten Beispiel vor dem Wert 44 und im zweiten Beispiel vor dem Wert 9999).

Aus demselben Grund lassen sich aufeinanderfolgende Zeilen einer Textdatei nur mit voneinander getrennten Aufrufen von *Readln* lesen. Wenn wir noch einmal von der oben gezeigten Beispieldatei ausgehen, dann liest ein Aufruf wie

```
Readln(f, String_var1, String_var2);
```

die gesamte Zeile (von »123« bis einschließlich »9999«, aber ohne den folgenden Zeilenvorschub) in die Variable *String_var1* ein. Danach ist das Zeilenende erreicht, das erst beim Abschluß von *Readln* übersprungen wird – *String_var2* bekommt folglich einen Nullstring zugewiesen.

Wenn *Readln* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus `{$I-}` den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Grenzen — *Readln* kann (im Gegensatz zu *Read*) nur auf Textdateien (inklusive der Standard-Eingabe) angewendet werden. Die entsprechende Datei muß für Leseoperationen eröffnet worden sein.

Querverweis — *Read, Write, Writeln*

Rectangle-Prozedur — Graph

Funktion — zeichnet ein Rechteck.

Deklaration — `Rectangle( x1, y1, x2, y2: Integer)`

Erläuterung — die Integer-Ausdrücke (*x1,y1*) legen die obere linke Ecke des zu zeichnenden Rechtecks fest, die Integer-Ausdrücke (*x2,y2*) die untere rechte Ecke. Für die Werte dieser vier Ausdrücke gelten die folgenden Beziehungen:

```
0 <= x1 < x2 <= GetMaxX
0 <= y1 < y2 <= GetMaxY
```

Etwas weniger technisch: alle vier Eckpunkte des sich ergebenden Rechtecks müssen innerhalb der Bildschirmgrenzen liegen, wobei die obere linke Ecke tatsächlich »links oberhalb« der unteren rechten Ecke liegen muß – einen automatischen Austausch von Wertepaaren (d.h. ein »Umdrehen« der Zeichenrichtung, falls notwendig) führt *Rectangle* nicht aus.

Rectangle verwendet die durch *SetLineStyle* gesetzte Linienart und Zeichenfarbe, wobei der jeweils letzte Aufruf von *SetWriteMode* bestimmt, ob der Zeichenvorgang direkt oder als XOR-Operation mit dem vorherigen Bildinhalt durchgeführt wird.

Grenzen vor dem Aufruf von *Rectangle* muß ein Grafikmodus gesetzt worden sein.

Querverweis *Bar, Bar3D, GetViewSettings, Line, SetColor, SetLineStyle, SetWriteMode.*

Beispiel

```
program RectangleDemo;
uses Crt, Graph;
var
  GraphMode, GraphDriver: Integer;
  x1,y1,Breite, Hoehe: Integer;
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
  Randomize;
  repeat
    SetColor(Random(16));
    Breite := Random(50); Hoehe := Random(40);
    x1:= Random(GetMaxX - Breite);
    y1:= Random(GetMaxY - Hoehe);
    Rectangle(x1,y1,x1+Breite,y1+Hoehe);
  until KeyPressed;
  Readln;
  CloseGraph;
end.
```

RegisterBGIDriver-Funktion — Graph

Funktion »registriert« einen als .OBJ-Datei eingebundenen Grafiktreiber.

Deklaration

```
RegisterBGIDriver(Driver: Pointer);
```

Ergebnistyp *Integer*

Erläuterung *RegisterBGIDriver* erwartet die Adresse eines als .OBJ-Datei direkt in das Programm eingebundenen Grafiktreibers und trägt diesen Treiber nach einer Prüfung in eine interne Tabelle ein. Auf diese Weise »registrierte« Treiber werden von *InitGraph* und *SetGraphMode* nicht mehr als .BGI-Datei auf der Diskette gesucht und auf den Heap geladen, sondern direkt verwendet.

(Die Umwandlung einer .BGI- in eine .OBJ-Datei geschieht mit dem Programm BINOBJ.EXE, das in Kapitel 10 »Die Zusatzprogramme« des *Benutzerhandbuches* besprochen ist).

Nach einer fehlerfreien Ausführung liefert *RegisterBGIDriver*

die (positive) Kennziffer des »registrierten« Grafiktreibers; Ergebnisse <0 zeigen einen Fehler an. (Da hier weder Diskettenoperationen noch die dynamische Speicherverwaltung mit im Spiel sind, gibt es praktisch nur die Möglichkeit einer zerstörten bzw. überholten Grafiktreiber-Kopfinformation, auf die *RegisterBGIDriver* mit dem Ergebnis -4 (Invalid driver) reagiert).

Im Prinzip ließe sich ein Grafiktreiber auch mit direkten Programmbefehlen auf den Heap laden und dann via *RegisterBGIDriver* »registrieren«. (Das Beispielprogramm im Abschnitt über *RegisterBGIFont* demonstriert diese Möglichkeit für Zeichensätze).

Hinweis: Die Adreßangabe bei *RegisterBGIDriver* hat eine Doppelfunktion: zum einen teilt sie dem Grafikpaket mit, daß der entsprechende Treiber direkt verfügbar ist; zum anderen hält sie den Linker davon ab, den Grafiktreiber als »unbenutzte Routine« komplett aus der .EXE-Datei hinauszuwerfen.

Grenzen — das »Registrieren« von Grafiktreibern muß *vor* dem Aufruf von *InitGraph* stattfinden.

Querverweis — *InitGraph*, *InstallUserDriver*, *RegisterBGIFont*.

Beispiel

```
program RegisterBGIDriverDemo;
uses Graph;
var
  GraphMode, GraphDriver: Integer;

{ Notwendige Schritte:
 1. Umwandlung des Treibers via BINOBJ
    in eine .OBJ-Datei:

    BINOBJ egavga.bgi  egavga.obj  egvga_driver
              ^          ^             ^
           Treiber     .OBJ-Datei  »Prozedurname«

 2. Definition einer »Prozedur« mit diesem Namen: }

procedure EGAVGA_Driver;  external;
{$L EGAVGA.OBJ}

{3. Registrierung dieser »Prozedur«
     mit RegisterBGIDriver}

begin
  if RegisterBGIDriver(@EGAVGA_Driver) < 0 then
  begin
    Writeln('Fehler beim "Registrieren" des ',
       'Grafiktreibers: ', GraphErrorMsg(GraphResult));
    Halt;
```

```
  end;

 GraphDriver := EGA; GraphMode := EGAHi;
 InitGraph(GraphDriver, GraphMode, '\TP');
 OutText('EGA-Grafik');
 LineTo(GetMaxX,GetMaxY);
 Readln;
 CloseGraph;
end.
```

RegisterBGIFont-Funktion — Graph

Funktion — »registriert« einen als .OBJ-Datei eingebundenen Vektor-Zeichensatz.

Deklaration — `RegisterBGIFont(Font: Pointer);`

Ergebnistyp — *Integer*

Erläuterung — *RegisterBGIFont* erwartet die Adresse eines als .OBJ-Datei direkt in das Programm eingebundenen Vektor-Zeichensatzes und trägt diesen Zeichensatz nach einer Prüfung in eine interne Tabelle ein. Auf diese Weise »registrierte« Zeichensätze werden von *InitGraph* und *SetGraphMode* nicht mehr als .CHR-Datei auf der Diskette gesucht und auf den Heap geladen, sondern direkt verwendet.

(Die Umwandlung einer .CHR- in eine .OBJ-Datei geschieht mit dem Programm BINOBJ.EXE, das in Kapitel 10 »Die Zusatzprogramme« des *Benutzerhandbuches* besprochen ist).

Nach einer fehlerfreien Ausführung liefert *RegisterBGIFont* die (positive) Kennziffer des »registrierten« Grafiktreibers; Ergebnisse <0 zeigen einen Fehler an:

Name	Wert	Beschreibung
grError	-11	kein Platz in der Zeichensatz-Tabelle
grInvalidFont	-13	Zeichensatz-Kopfinformation ungültig
grInvalidFontNum	-14	Zeichensatz-Nummer in der Kopfinformation unbekannt

(Die Fehlerbedingung *grError* ist allerdings recht unwahrscheinlich: das Grafikpaket bietet Platz für 10 verschiedene Zeichensätze – und nur vier davon sind mit *GothicFont..SansSerifFont* besetzt).

Im Prinzip läßt sich ein Grafiktreiber auch mit direkten Programmbefehlen auf den Heap laden und dann via *RegisterBGIFont* »registrieren« (siehe »Beispiel«).

Hinweis: Die Adreßangabe bei *RegisterBGIFont* hat eine Doppelfunktion: zum einen teilt sie dem Grafikpaket mit, daß der

entsprechende Zeichensatz direkt verfügbar ist; zum anderen hält sie den Linker davon ab, den Zeichensatz als »unbenutzte Routine« komplett aus der .EXE-Datei hinauszuwerfen.

Querverweis *InitGraph, InstallUserFont, RegisterBGIDriver.*

Beispiel

```
program RegisterBGIFontDemo;
uses Graph;

var  GraphDriver, GraphMode, x: Integer;
     FontF: file;
     FontP: Pointer;

begin  { Laden des Zeichensatzes "TriplexFont" }
  Assign(FontF, 'TRIP.CHR'); Reset(FontF, 1);
  GetMem(FontP, FileSize(FontF));
  BlockRead(FontF, FontP^, FileSize(FontF));
  Close(FontF);

  { "Registrieren" des Zeichensatzes }
  if RegisterBGIFont(FontP) < 0 then
  begin
    Writeln('Fehler beim Registrieren des ',
      'Zeichensatzes: ', GraphErrorMsg(GraphResult));
    Halt;
  end;

  { Start des Grafikpakets }
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');

  { Diese beiden Zeichensätze sind nicht "registriert"
    - deshalb müssen sie jedesmal wieder von der
    Diskette geladen werden: }
  for x:= 1 to 10 do
  begin
    SetTextStyle(GothicFont, HorizDir, 4);
    OutText('G');
    SetTextStyle(SansSerifFont, HorizDir, 4);
    OutText('S');
  end;
  Readln;
  { Hier ist einer der beiden Zeichensätze
    "registriert" - folglich kann der andere
    auf dem Heap bleiben; die Diskettenzugriffe
    nach dem ersten Laden entfallen. }
  MoveTo(0,100);
  for x:= 1 to 10 do
  begin
    SetTextStyle(TriplexFont, HorizDir, 4);
    OutText('T');
```

```
    SetTextStyle(GothicFont, HorizDir, 4);
    OutText('G');
  end;
  Readln;
  CloseGraph;
end.
```

Release-Prozedur

Funktion — setzt den Heap auf einen zuvor mit *Mark* festgehaltenen Zustand zurück.

Deklaration — `Release(var p:Pointer)`

Erläuterung — *p* ist eine Zeigervariable beliebigen Typs, der zu einem vorherigen Zeitpunkt mit *Mark* ein Wert zugewiesen wurde. Der Aufruf *Release(p)* löscht alle dynamischen Variablen, die seit dem Aufruf *Mark(p)* über *New* oder *GetMem* erzeugt wurden.

Vorsicht: Vom Grafikpaket geladene Treiber und Zeichensätze unterscheiden sich in keiner Weise von anderen dynamischen Variablen – mit einer Ausnahme: die versehentliche »Freigabe« eines Grafiktreibers mit *Mark/Release* bringt das System mit Sicherheit zum Absturz.

Eine ausführliche Besprechung der zugrundeliegenden Mechanismen finden Sie in Kapitel 14 »Interne Details« des *Programmierhandbuches*.

Grenzen — *Mark/Release* und *Dispose/FreeMem* können nur dann in gemischter Form verwendet werden, wenn sich der Programmierer dabei strikt an bestimmte Regeln hält (siehe Kapitel 14 »Interne Details« des *Programmierhandbuches*).

Querverweis — *Mark*, *Dispose*, *FreeMem*

Rename-Prozedur

Funktion — gibt einer externen Diskettendatei einen neuen Namen.

Deklaration — `Rename(var f; newname: string)`

Erläuterung — *f* ist eine Datei-Variable beliebigen Typs, die über *Assign* mit einer externen Diskettendatei verbunden wurde. Diese Datei darf zum Zeitpunkt des Aufrufs von *Rename* nicht offen sein.

newname ist ein String-Ausdruck und gibt den neuen Namen der Datei an. Weitere Operationen über *f* werden von *Rename* nicht beeinflußt.

Hinweis: Im Gegensatz zum DOS-Befehl *rename* kann die Prozedur *Rename* Dateien auch zwischen verschiedenen Verzeichnissen bewegen. Eine Befehlsfolge wie

```
Assign(f,'\TP\GRAFIK\RENTEST.PAS');
Rename(f, '\TP\RENAME.PAS');
```

versetzt also die im Verzeichnis \TP\GRAFIK gespeicherte Datei RENTEST.PAS in das Verzeichnis \TP und gibt ihr den Namen RENAME.PAS.

Achtung: Kopieraktionen von Dateiinhalten werden von *Rename* nicht ausgeführt – diese Prozedur versetzt und verändert lediglich Datei*einträge*. Der Versuch, eine Datei von einem Laufwerk auf ein anderes »umzubenennen«, endet mit einem Laufzeitfehler.

Wenn *Rename* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus `{$I-}` den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Grenzen *Rename* darf unter keinen Umständen auf eine offene Datei angewendet werden.

Querverweis *Erase*

Reset-Prozedur

Funktion öffnet eine existierende Datei.

Deklaration `Reset(var f [: file;  recsize: Word] )`

Erläuterung *f* steht für eine Datei-Variable beliebigen Typs, der zuvor mit *Assign* eine externe Datei zugeordnet wurde. Der optionale Parameter *recsize* ist ein Ausdruck des Typs *Word*, legt die Recordgröße fest und kann nur auf untypisierte Dateien angewendet werden.

Reset öffnet die Datei mit dem bei *Assign* angegebenen Namen bzw. bricht mit einem Laufzeitfehler ab, wenn keine Datei des angegebenen Namens existiert (*Assign* selbst nimmt keine Prüfungen vor). Ist die durch *f* bezeichnete Datei zum Zeitpunkt des Aufrufs von *Reset* bereits offen, wird sie zuerst geschlossen und danach wieder eröffnet. In beiden Fällen setzt *Reset* den Positionszeiger innerhalb dieser Datei auf den Dateianfang.

Wenn *f* mit *Assign* ein Nullstring als Name zugeordnet wurde (*Assign(f, '')*), dann bezieht sich *f* nach dem Aufruf von *Reset* auf die Standard-Eingabe (*Input*, DOS-Handle 0).

Wenn *f* als Textdatei-Variable deklariert ist, eröffnet *Reset* die damit verbundene Datei ausschließlich für Leseoperationen. Falls die Datei leer ist (0 Bytes Umfang), dann liefert *Eof(f)* bereits unmittelbar nach der Eröffnung mit *Reset* den Wert *True* zurück; ansonsten ist das Ergebnis dieser Funktion *False*.

Wenn *f* als untypisierte Datei-Variable deklariert ist, dann öffnet *Reset* die damit verbundene Datei für Lese- und Schreib-

operationen, die Recordgröße der Datei kann über den optionalen Parameter *recsize* bei der Eröffnung festgelegt werden. Ist dieser Parameter nicht angegeben, dann setzt *Reset* eine Recordlänge von 128 als Standardvorgabe.

Wenn *Reset* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Querverweis *Append*, *Assign*, *Rewrite*, *Truncate*

Beispiel

```
program ResetDemo;
function Datei_vorhanden (Dateiname: string) : Boolean;
var f: file;
begin
  Assign(f, Dateiname);
  {$I-} Reset(f); {$I+}
  if IOResult = 0 then begin
     Close(f);
     Datei_vorhanden := Dateiname <> '';
   end else
     Datei_vorhanden := False;
end;

begin
  { Dateiname wird als 1. Kommandozeilen-
    Parameter angenommen }

  write('Datei existiert');
  if not Datei_vorhanden(ParamStr(1))
    then write(' nicht!');
end.
```

RestoreCrt-Prozedur — Crt

Funktion setzt wieder den Videomodus, der beim Start des Programms aktiv war.

Deklaration `RestoreCrtMode`

Erläuterung *RestoreCrtMode* verwendet einen BIOS-Aufruf, um den Videomodus wieder zu setzen, der beim Start des Programms aktiv war. Bei Verwendung dieser Prozedur kann sich ein Programm selbst dann »ordentlich« verabschieden, wenn *Crt* diesen Original-Videomodus nicht direkt unterstützt.

Grenzen nach einem Aufruf von *RestoreCrtMode* ist der Videomodus für die Routinen von *Crt* in einem »undefinierten« Zustand. *RestoreCrt* sollte deshalb der letzte Schritt eines Programms vor dem Rücksprung zur DOS-Kommandoebene sein.

Querverweis *TextMode*

RestoreCrtMode-Prozedur — Graph

Funktion setzt wieder den Videomodus, der vor dem Start des Grafikpakets aktiv war.

Deklaration
```
RestoreCrtMode
```

Erläuterung *InitGraph* hält beim Start des Grafikpakets den ursprünglich aktiven Videomodus fest; mit *RestoreCrtMode* und *SetGraphMode* sind deshalb beliebige Umschaltungen zwischen »Text« und »Grafik« möglich.

Hinweis: Der Aufruf der Prozedur *TextMode* mit der Variablen *LastMode* als Parameter hat dieselbe Wirkung wie *RestoreCrtMode*.

Grenzen vor dem Aufruf von *RestoreCrtMode* muß das Grafikpaket mit *InitGraph* aktiviert worden sein – ansonsten ist der »vorherige« Videomodus undefiniert.

Querverweis *DetectGraph*, *InitGraph*, *SetGraphMode*, *Textmode*.

Beispiel
```
.... { Grafik setzen }
RestoreCrtMode;   { Umschaltung auf Text }
....
SetGraphMode(EGA); { zur¡ck in die Grafik }
```

Rewrite-Prozedur

Funktion erzeugt und eröffnet eine neue Datei.

Deklaration
```
Rewrite(var f [: file  ; recsize: Word] )
```

Erläuterung *f* steht für eine Datei-Variable beliebigen Typs, der zuvor mit *Assign* eine externe Datei zugeordnet wurde. Der optionale Parameter *recsize* ist ein Ausdruck des Typs *Word*, legt die Recordgröße fest und kann nur auf untypisierte Dateien angewendet werden.

Rewrite erzeugt eine Datei mit dem bei *Assign* angegebenen Namen. Wenn die durch *f* bezeichnete Datei zum Zeitpunkt des Aufrufs von *Rewrite* bereits existiert, wird sie vollständig gelöscht und danach neu erzeugt. Dasselbe ist der Fall, wenn die Datei zum Zeitpunkt des Aufrufs von *Rewrite* offen ist. In allen drei Fällen setzt *Reset* den Positionszeiger innerhalb dieser Datei auf den Dateianfang.

Wenn *f* mit *Assign* ein Nullstring als Name zugeordnet wurde (*Assign*(*f*, '')), dann bezieht sich *f* nach dem Aufruf von *Rewrite* auf die Standard-Ausgabe (*Output*, DOS-Handle 1).

Wenn *f* als Textdatei-Variable deklariert ist, eröffnet *Rewrite*

die damit verbundene Datei ausschließlich für Schreiboperationen.

Wenn *f* als untypisierte Datei-Variable deklariert ist, dann öffnet *Rewrite* die damit verbundene Datei für Lese- und Schreiboperationen, die Recordgröße der Datei kann über den optionalen Parameter *recsize* bei der Eröffnung festgelegt werden. Ist dieser Parameter nicht angegeben, dann setzt *Reset* eine Recordlänge von 128 als Standardvorgabe.

Wenn *Rewrite* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus `{$I-}` den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Querverweis *Append*, *Assign*, *Reset*, *Truncate*

Beispiel

```
var
  f : Text;

begin
  Assign(f, 'NEUTEXT.TXT');
  Rewrite(f);
  Writeln(f, 'Gerade eine neue Datei mit einem ',
             'recht nichtssagenden Inhalt erzeugt...');
  Close(f);
end.
```

RmDir-Prozedur

Funktion löscht ein leeres Unterverzeichnis

Deklaration `RmDir(s: string)`

Erläuterung *s* ist ein String-Ausdruck, gibt den Namen des zu löschenden Unterverzeichnisses an und kann optional einen Laufwerksbezeichner und/oder einen Suchweg enthalten.

Wenn der angegebene Suchweg nicht existiert, das zu löschende Unterverzeichnis noch Dateieinträge enthält oder momentan als Standard-Verzeichnis gesetzt ist, dann erzeugt *RmDir* einen Laufzeitfehler.

Wenn *RmDir* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus `{$I-}` den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Querverweis *ChDir*, *GetDir*, *MkDir*

Beispiel

```
program RmDirDemo;
begin
  { Name des Unterverzeichnisses wird als
    1. Kommandozeilen-Parameter vorausgesetzt }
```

```
  {$I-} RmDir(ParamStr(1)); {$I+}
  if IOResult = 0 then
    writeln('Unterverzeichnis gelöscht!')
   else
    writeln('... kann nicht gelöscht werden.');
end.
```

Round-Funktion

Funktion — rundet das (Real-)Argument auf einen ganzzahligen Wert.

Deklaration — `Round(x:Real);`

Ergebnistyp — *LongInt*

Erläuterung — *x* ist ein Real-Ausdruck, der von *Round* auf den nächstliegenden ganzzahligen Wert gerundet und danach in einen *LongInt* konvertiert wird.

Befindet sich der Wert von *x* exakt zwischen zwei ganzen Zahlen, wird in Richtung des größeren Absolutbetrags gerundet: *Round*(*-3.5*) ergibt also *-4*, *Round*(*3.5*) ergibt *4*. Wenn der Wert von *x* außerhalb des zulässigen Rechenbereichs von *LongInt* liegt, erzeugt *Round* einen Laufzeitfehler.

Querverweis — *Int*, *Trunc*

RunError-Prozedur

Funktion — erzeugt einen Laufzeitfehler, der das Programm definiert abbricht.

Deklaration — `RunError [(ErrorCode: Word) ]`

Erläuterung — *RunError* entspricht in seiner Wirkungsweise der Standardprozedur *Halt* – abgesehen davon, daß diese Routine zusätzlich die Ausgabe einer Laufzeitfehlermeldung veranlaßt (wobei die Adresse des Punktes angegeben wird, von dem aus *RunError* aufgerufen wurde).

Der Integer-Ausdruck *ErrorCode* ist optional und legt die auszugebende Fehlernummer fest – wenn er fehlt, setzt der Compiler hier den Wert 0 ein.

Diese Routine ist vor allem für das Austesten von Exit-Prozeduren gedacht, die ein Programm im Fehlerfall »ordentlich« abschließen sollen.

Hinweis: Wenn das Modul, das den Aufruf von *RunError* enthält, mit *Debug information..On* compiliert wurde, reagiert die integrierte Entwicklungsumgebung exakt so, als ob ein echter Laufzeitfehler stattgefunden hätte (d.h. mit der Lokali-

sierung der »Fehlerstelle« und der Ausgabe einer entsprechenden Meldung).

Querverweis *Halt, Exit*

Beispiel

```
....
{$IFDEF DEBUG}   { <- Ein Symbol, das für den Test des
                          Programms definiert wurde }
  if P = nil then RunError(204);
{$ENDIF}
```

Sector-Prozedur Graph

Funktion zeichnet ein ausgefülltes elliptisches »Kuchenstück«.

Deklaration

```
Sector(x,y: Integer;
       StAngle, EndAngle, XRadius, YRadius: Word)
```

Erläuterung *Sector* erfüllt dieselbe Funktion für Ellipsen wie *PieSlice* für Kreise: Diese Routine zeichnet ein Kreisbogenstück, dessen Mittelpunkt durch die Integer-Ausdrücke (*x,y*) angegeben ist.

StAngle legt den Startpunkt des mit *XRadius* (horizontal) und *YRadius* (vertikal) gezogenen Kreisbogens fest, *EndAngle* sein Ende. Die Angabe beider Winkel erfolgt in *Grad*, wobei die Zählung entgegen dem Uhrzeigersinn läuft (siehe *Arc*-Prozedur).

Die Endpunkte des Kreisbogens werden mit dem Kreismittelpunkt verbunden, danach füllt *Sector* das entstandene »Kuchenstück« mit dem durch *SetFillPattern* bzw. *SetFillStyle* festgelegten Muster. Wenn während des Ausfüllens ein Fehler auftritt, liefert ein nachfolgender Aufruf von *GraphResult* den Wert -6.

Für das Zeichnen des Kreisbogens wird die momentane Zeichenfarbe und die mit *SetLineStyle* gesetzte Linienart verwendet.

Der Aufruf von *Sector* mit den Werten 0 und 360 für *StAngle* bzw. *EndAngle* zeichnet einen vollständigen ausgefüllten Kreis, entspricht also der Prozedur *FillEllipse*.

Hinweis: Jeder Grafiktreiber (*.BGI) enthält einen Wert zur Korrektur der physikalischen Höhen-/Seitenverhältnisse des Bildschirms (*AspectRatio*), der automatisch von *Arc*, *Circle*, *PieSlice*, *Sector* usw. verwendet wird, um auch optisch runde Kreise zu erhalten.

Grenzen vor dem Aufruf von *Sector* muß ein Grafikmodus gesetzt worden sein.

Querverweis *Arc*, *Circle*, *Ellipse*, *GetArcCoords*, *GetAspectRatio*, *PieSlice*, *SetAspectRatio*, *SetFillStyle*, *SetFillPattern*, *SetLineStyle*, *SetGraphBufSize*.

Beispiel

```
program SectorDemo;
uses Graph;
var
  GraphDriver, GraphMode: Integer;
  x : Integer;
begin
  GraphDriver := Detect;   { automatische Erkennung }
  InitGraph(GraphDriver, GraphMode, '\TP');

  for x := 10 downto 1 do
  begin
   SetFillStyle(WideDotFill, x);
   Sector(GetMaxX - 10, GetMaxY div 2, 120 - x, 240 - x,
          x * GetMaxX div 10 - 20, x * GetMaxY div 20);

   end;
  Readln;
  CloseGraph;
end.
```

Seek-Prozedur

Funktion — setzt den Positionszeiger in einer Datei auf eine bestimmte Komponente.

Deklaration — `Seek(var f; n: LongInt)`

Erläuterung — *f* steht für eine Datei-Variable beliebigen Typs (außer *Text*), *n* ist ein Ausdruck vom Typ *LongInt* und gibt die Nummer der Komponente an, auf die der Positionszeiger innerhalb der Datei gesetzt werden soll.

Die Zählung der Komponenten einer Datei beginnt mit 0 – *Seek(f,0)* setzt den Positionszeiger also an den Dateianfang. Damit sich eine Datei durch nachfolgende Schreiboperationen vergrößern läßt, ist der Aufruf *Seek(f,FileSize(f))* definiert: er setzt den Positionszeiger *hinter* die letzte Komponente.

Wenn *Seek* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus `{$I-}`den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Grenzen — *Seek* setzt voraus, daß die durch *f* bezeichnete Datei offen ist, und kann nicht auf Textdateien angewendet werden.

Querverweis — *FilePos*

SeekEof-Funktion

Funktion — prüft, ob sich zwischen der momentanen Position und dem Ende einer Datei noch »lesbare» Daten befinden.

Deklaration — `SeekEof [ (var f: Text) ]`

Ergebnistyp — *Boolean*

Erläuterung — *SeekEof* folgt denselben Regeln wie *Eof* – mit einem Unterschied: *Eof* bewegt den Positionszeiger innerhalb der Datei nicht und liefert nur dann *True*, wenn das Dateiende bereits vor dem Aufruf der Funktion erreicht wurde. *SeekEof* liest dagegen so lange Zeichen ab der momentanen Position, bis entweder ein »lesbares« Zeichen (kein Leerzeichen, Tab oder Zeilenvorschub) oder das Dateiende erreicht ist – erst dann wird die eigentliche Prüfung ausgeführt und ein entsprechendes Ergebnis zurückgeliefert. Sinnvoll ist *SeekEof* vor allem beim Lesen numerischer Daten.

Wenn *SeekEof* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus `{$I-}` den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Grenzen — *SeekEof* setzt voraus, daß *f* eine offene Textdatei bezeichnet.

Querverweis — *Eof*, *SeekEoln*

SeekEoln-Funktion

Funktion — prüft, ob sich zwischen der momentanen Position und dem nächsten Zeilenende einer Datei noch »lesbare» Daten befinden.

Deklaration — `SeekEoln [ (var f: Text) ]`

Ergebnistyp — *Boolean*

Erläuterung — *SeekEoln* folgt denselben Regeln wie *Eoln* – mit einem Unterschied: *Eoln* bewegt den Positionszeiger innerhalb der Datei nicht und liefert nur dann *True*, wenn ein Zeilenende bereits vor dem Aufruf der Funktion erreicht wurde. *SeekEoln* liest dagegen solange Zeichen ab der momentanen Position, bis entweder ein »lesbares« Zeichen (kein Leerzeichen oder Tab) oder ein Zeilenende (bzw. das Dateiende) erreicht ist – erst dann wird die eigentliche Prüfung ausgeführt und ein entsprechendes Ergebnis zurückgeliefert. Sinnvoll ist *SeekEoln* vor allem beim Lesen von numerischen Daten.

Wenn *SeekEoln* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus `{$I-}`

den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Grenzen *SeekEoln* setzt voraus, daß *f* eine offene Textdatei bezeichnet.

Querverweis *Eoln*, *SeekEof*

Seg-Funktion

Funktion liefert die Segmentadresse des angegebenen Objekts zurück.

Deklaration `Seg(x)`

Ergebnistyp *Word*

Erläuterung *x* steht für eine beliebige Variable, einen beliebigen Variablenbezug oder den Bezeichner einer Prozedur oder Funktion. Das Ergebnis von *Seg* ist die Adresse des Segments, in dem *x* gespeichert ist.

Querverweis *Addr*, *Ofs*

SetActivePage-Prozedur Graph

Funktion legt fest, auf welcher Grafik-Seite folgende Zeichenbefehle arbeiten.

Deklaration `SetActivePage(Page: Word)`

Erläuterung mehrere Grafikseiten werden nur von EGA- und VGA-Karten sowie von der Hercules-Karte unterstützt. Mit Video-Adaptern dieser Typen ist es möglich, dem Benutzer eine Grafik-Seite zu zeigen, während »im Hintergrund« auf eine andere Seite gezeichnet wird.

Hinweis: Die Umschaltung zwischen der *Darstellung* mehrerer Grafikseiten erfolgt mit der Prozedur *SetVisualPage*.

Grenzen vor dem Aufruf von SetActivePage muß ein Grafikmodus gesetzt worden sein.

Querverweis *SetVisualPage*.

Beispiel

```
program SetVisualPageDemo;
uses Graph;
var
  GraphDriver, GraphMode: Integer;

begin
  GraphDriver := Detect;  { automatische Erkennung }
  InitGraph(GraphDriver, GraphMode, '\TP');

  ClearDevice;
  SetVisualPage(0);     { erste Grafik-Seite sichtbar }
```

```
  OutText('Bitte warten....');     { geht auf Seite 0 }
  SetActivePage(1);
  Line(10,10,200,200);   { Zeichnen im "Hintergrund" }

  SetVisualPage(1);      { zack! völlig überraschend
                           erscheint das heimlich
                           erstellte Kunstwerk! }
  Readln;
  CloseGraph;
end.
```

SetAllPalette-Prozedur — Graph

Funktion — setzt sämtliche Einträge einer Farb-Palette neu.

Deklaration

```
SetAllPalette(var Palette)
```

Erläuterung — *Palette* ist ein untypisierter Parameter, der die folgende Struktur haben muß:

```
Anzahl : Byte;   { Anzahl der folgenden Einträge }
Farben : array[0..Anzahl-1] of ShortInt;
```

Jedes Element des Arrays *Farben* ersetzt einen Eintrag der Farb-Palette und kann Werte im Bereich von -1..63 haben. (Der »Farbwert« -1 bedeutet, daß der korrespondierende Eintrag der Farb-Palette unverändert bleiben soll).

Hinweis: Der Wert -1 steht unabhängig vom verwendeten Grafiktreiber und -modus immer für »keine Veränderung« – ganz im Gegensatz zu den Werten, die als tatsächliche Einträge der Farb-Palette erlaubt sind.

Die folgenden Konstanten und Datentypen sind in der Unit *Graph* für die Arbeit mit Farb-Paletten definiert:

```
const
  Black        = 0;   { Schwarz }
  Blue         = 1;   { Blau }
  Green        = 2;   { Grün }
  Cyan         = 3;   { Türkis }
  Red          = 4;   { Rot }
  Magenta      = 5;   { Fuchsinrot }
  Brown        = 6;   { Braun }
  LightGray    = 7;   { Hellgrau }
  DarkGray     = 8;   { Dunkelgrau }
  LightBlue    = 9;   { Hellblau }
  LightGreen   = 10;  { Hellgrün }
  LightCyan    = 11;  { helles Türkis }
  LightRed     = 12;  { Hellrot }
  LightMagenta = 13;  { helles Fuchsinrot }
```

```
  Yellow       = 14;  { Gelb }
  White        = 15;  { Weiß }

MaxColors = 15;  { maximal 16 Paletten-Einträge }

type
  PaletteType = record
      Size   : Byte;
      Colors : array[0..MaxColors] of ShortInt;
    end;
```

Veränderungen innerhalb der Farb-Palette werden sofort auf dem Bildschirm sichtbar. Das im »Beispiel« gezeigte Programm zeichnet einige Balken in verschiedenen Farben und ändert danach die entsprechenden Einträge der Farb-Palette.

Grenzen

vor dem Aufruf von *SetAllPalette* muß ein Grafikmodus gesetzt worden sein. Diese Routine läßt sich nicht zusammen mit dem Grafiktreiber IBM8514.BGI sowie dem VGA-Modus mit 256 Farben verwenden.

Querverweis

GetBkcolor, GetColor, GetPalette, GraphResult, SetBkColor, SetColor, SetPalette, SetRGBPalette.

Beispiel

```
program  SetAllPaletteDemo;
uses Crt,  Graph;
var
  GraphMode, GraphDriver: Integer;
  Palette: PaletteType;
  x, y : Integer;

begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP')

  for x:= 1 to 9 do begin
    SetFillStyle(x, x mod 4 + 1);
    Bar(x * 60, 20, (x+1) * 60, 200);
  end;
  GetPalette(Palette);
  repeat
    with Palette do begin
      Size:= 5;
      Colors[0] := 0;
      for y := 1 to 4 do
        Colors[y] := Random(64);
      end; { with Palette }
      SetAllPalette(Palette);
      Delay(600);
   until KeyPressed;
  Readln;
```

```
  CloseGraph;
end.
```

SetAspectRatio-Prozedur — Graph

Funktion — setzt den Korrekturfaktor für das Höhen-/Seitenverhältnis des Bildschirms direkt.

Deklaration

```
SetAspectRatio(XAsp , YAsp: Word)
```

Erläuterung — jedem Grafiktreiber und jedem Grafikmodus ist ein *Aspekt* zugeordnet, der von Prozeduren wie *Arc*, *Circle*, *Ellipse* usw. automatisch verwendet wird, damit Kreise und Ellipsen auch optisch rund erscheinen.

Falls Sie das nicht tun (weil Sie z.B. eine unübliche Kombination von Video-Adapter und Monitor verwenden), gibt es zwei Möglichkeiten zur Korrektur: entweder ein Nachstellen des Monitors oder eine direkte Veränderung des von Turbo Pascal verwendeten Faktors.

Zu letzterem dient die Prozedur *SetAspectRatio* – sie arbeitet mit denselben Wertepaaren, die von *GetAspectRatio* zurückgeliefert werden: *YAsp* hat den konstanten Wert 10000; *XAsp* steht im entsprechenden Verhältnis dazu.

Grenzen — vor dem Aufruf von *SetAspectRatio* muß ein Grafikmodus gesetzt worden sein. Beim Wechsel des Grafiktreibers geht der direkt gesetzte Wert verloren.

Querverweis — *GetAspectRatio*.

Beispiel

```
program SetAspectRatioDemo;
uses Crt, Graph;
var
  GraphDriver, GraphMode: Integer;
  XAsp, YAsp: Word;
{ Gibt die momentanen Werte von XAsp & YAsp aus }
procedure PrintXYAsp(YPos: Integer);
var  S1, S2: String;
begin
  SetViewPort(0,YPos,200,YPos+20,False);{ L´schen des }
  ClearViewPort;                        { Ausgabebereichs }
  SetViewPort(0,0,GetMaxX, GetMaxY, False);
  Str(XAsp, S1); Str(YAsp, S2);
  OutTextXY(0,YPos, 'XAsp: '+ S1 + ' YAsp: ' + S2);
end;

begin
  GraphDriver := Detect;   { automatische Erkennung }
  InitGraph(GraphDriver, GraphMode, '\TP');

  GetAspectRatio(XAsp, YAsp); { Standardwerte
```

```
                                    ermitteln }
  PrintXYAsp(20);                 { Ausgabe gründlich }
  XAsp := YAsp div 5;             { "verbiegen" }

  while (XAsp <= YAsp) and not KeyPressed do
  begin
    SetAspectRatio(XAsp, YAsp);
    Circle(GetMaxX div 2, GetMaxY div 2, 100);
    PrintXYAsp(40);
    Inc(XAsp, 20);
  end;
  Readln;
  CloseGraph;
end.
```

SetBkColor-Prozedur — Graph

Funktion — setzt die Hintergrundfarbe des Bildschirms.

Deklaration

```
SetBkColor(ColorNum: Word)
```

Erläuterung — *ColorNum* ist ein Ausdruck des Typs *Word* und legt fest, welcher Paletten-Eintrag künftig als Hintergrundfarbe benutzt werden soll.

Um die Hintergrundfarbe des Bildschirms zu verändern, gibt es zwei Möglichkeiten: entweder eine Änderung des ersten Paletten-Eintrags (*Palette(0)*) oder die Festlegung, welcher Paletten-Eintrag als Hintergrund benutzt werden soll.

SetBkColor benutzt die zweite Möglichkeit: der Aufruf *SetBkColor(0)* legt den ersten Paletten-Eintrag als Hintergrund fest, der Aufruf *SetBkcolor(1)* den zweiten Eintrag usw. (Natürlich lassen sich auch diese Paletten-Einträge selbst noch verändern).

Grenzen — vor dem Aufruf von *SetBkColor* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *GetBkColor, GetColor, GetPalette, SetAllPalette, SetColor, SetPalette, SetRGBPalette.*

Beispiel

```
program SetBkColorDemo;
uses Crt, Graph;
var
  GraphMode, GraphDriver: Integer;
  Palette : PaletteType;
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
  Randomize;
  GetPalette(Palette);
```

```
  repeat
    SetBkColor(Random(Palette.Size));
    LineTo(Random(GetMaxX), Random(GetMaxY));
    Delay(100);
  until KeyPressed;
  Readln;
  CloseGraph;
end.
```

SetCBreak-Prozedur — Dos

Funktion — legt fest, bei welchen Operationen DOS auf Ctrl Break prüft.

Deklaration — `SetCBReak(Break: Boolean)`

Erläuterung — *SetCBreak* setzt das *Break*-Flag von DOS (über die DOS-Funktion $33) mit dem Wert des als *Break* übergebenen Ausdrucks:

True »Überprüfung an« – DOS prüft bei jedem Funktionsaufruf, ob Ctrl Break gedrückt wurde;

False »Überprüfung aus« – die Prüfung auf Ctrl Break findet nur bei Ein- und Ausgaben über Tastatur, Bildschirm, serielle Schnittstellen und Drucker statt.

Hinweis: Mit der Prozedur *GetCBreak* läßt sich der momentane Wert des *Break*-Flags ermitteln.

Querverweis — *GetCBreak*

SetColor-Prozedur — Graph

Funktion — legt fest, welcher Paletten-Eintrag als Zeichenfarbe benutzt werden soll.

Deklaration — `SetColor(Color: Word)`

Erläuterung — *Color* ist ein Ausdruck des Typs *Word* und legt die Zeichenfarbe indirekt fest: der Aufruf *SetColor(0)* bestimmt den ersten Paletten-Eintrag als Zeichenfarbe, der Aufruf *SetColor(1)* den zweiten Eintrag usw. Welche physikalische Farbe dabei herauskommt, hängt vom Wert des jeweils gewählten Paletten-Eintrags ab.

Ein Beispiel: im Modus CGAC1 enthält die Farb-Palette vier Einträge: Schwarz, Rot, Gelb und Grün. *SetColor(0)* setzt Schwarz als Zeichenfarbe, *SetColor(1)* setzt Rot usw.

Grenzen — vor dem Aufruf von *SetColor* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *GetBkColor*, *GetColor*, *GetPalette*, *SetAllPalette*, *SetBkColor*, *SetPalette*, *SetRGBPalette*.

Beispiel

```
program SetColorDemo;
usesCrt, Graph;
var
  GraphMode, GraphDriver: Integer;
  Palette : PaletteType;
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
  Randomize;
  GetPalette(Palette);
  repeat
    SetColor(Random(Palette.Size));
    LineTo(Random(GetMaxX), Random(GetMaxY));
  until KeyPressed;
  Readln;
  CloseGraph;
end.
```

SetDate-Prozedur — Dos

Funktion — setzt das Kalenderdatum des Betriebssystems.

Deklaration

```
SetDate(Year , Month, Day, DayofWeek: Word)
```

Erläuterung — gültige Werte für *Year* gehen von 1980..2099, für *Month* von 1..12, für *Day* von 1..31 und für *DayofWeek* (Wochentag) von 0..6, wobei 0 für den Sonntag steht.

Querverweis — *GetTime, GetDate, SetTime*

SetFAttr-Prozedur — Dos

Funktion — setzt die Attribute einer Datei.

Deklaration

```
SetFAttr(var f; Attr: Byte)
```

Erläuterung — *f* steht für eine Datei-Variable beliebigen Typs, der über *Assign* der Name einer externen Diskettendatei zugeordnet wurde. Die Datei darf nicht offen sein.

Attr ist ein Ausdruck des Typs *Byte*, dessen Wert sich aus der Addition mehrerer in der Unit *Dos* definierter Konstanten ergibt:

```
const
  ReadOnly  = $01;
  Hidden    = $02;
  SysFile   = $04;
  VolumeID  = $08;
  Directory = $10;
```

```
  Archive   = $20;
  AnyFile   = $3F;
```

Fehler während der Ausführung von *SetFAttr* setzen die globale Variable *DosError*, mögliche Codes sind hier 3 (ungültiger Suchweg) und 5 (Dateizugriff verweigert).

Grenzen — die durch *f* bezeichnete Datei darf nicht offen sein.

Querverweis — *GetFAttr*, *GetFTime*, *SetFTime*

Beispiel

```
program SetFAttrDemo;
  { Quelltext: SETFATT.PAS }
uses Dos;
var
  f : file;
begin
  Assign(f, 'SETFATT.PAS');
  SetFAttr(f, ReadOnly + Hidden + SysFile);
  Writeln('So. Und jetzt dürfen Sie mal versuchen,');
  Writeln('diese Datei wieder zu löschen.');
end.
```

SetFillPattern-Prozedur — Graph

Funktion — ermöglicht die freie Definition von Mustern für Flächenfüllungen.

Deklaration

```
SetFillPattern(Pattern: FillPatternType; Color: Word)
```

Erläuterung — *SetFillPattern* legt das Muster und die Farbe für nachfolgende Aufrufe der Prozeduren *FillPoly*, *FloodFill*, *Bar*, *Bar3D*, *PieSlice* und *Sector* fest. *Color* ist ein Ausdruck des Typs *Word* und bestimmt die Farbe des Musters; *Pattern* definiert das Muster und hat den Typ *FillPatternType*, der in der Unit *Graph* folgendermaßen definiert ist:

```
type
  FillPatternType = array[1..8] of Byte;
```

Jedes Bit dieses Arrays steht für ein Pixel: wenn es auf »1« gesetzt ist, wird ein Punkt in der durch *Color* gesetzten Farbe gezeichnet; ist es auf »0« gesetzt, bleibt die entsprechende Stelle des Bildschirms leer.

Jedes Byte legt acht nebeneinanderliegende Punkte einer Bildschirmzeile fest, die Bytes selbst sind »untereinander« angeordnet; ein Schachbrettmuster mit der Graustufe 0.5 ist beispielsweise folgendermaßen definiert:

```
Binär       Hex
10101010    $AA     (1. Byte)
01010101    $55     (2. Byte)
```

```
10101010      $AA     (3. Byte)
01010101      $55     (4. Byte)
10101010      $AA     (5. Byte)
01010101      $55     (6. Byte)
10101010      $AA     (7. Byte)
01010101      $55     (8. Byte)
```

Das Grafikpaket erlaubt jeweils nur ein »aktives« Muster, das entweder durch *SetFillPattern* oder durch *SetFillStyle* gesetzt und von allen folgenden Flächenfüllungen benutzt wird.

Im Prinzip führt *SetFillPattern* zwei aufeinanderfolgende Aktionen aus: zuerst die Definition des Musters und dann einen Aufruf von *SetFillStyle* mit den Parametern *UserFill* und *Color*. Aus diesem Grund läßt sich ein Ablauf wie der folgende konstruieren:

```
SetFillPattern(x, Green); {benutzerdefiniertes
                           Muster X}
{ ... Zeichenoperationen }
SetFillStyle(SolidFill, Blue);  {vorgegebenes Muster}
{ ... Zeichenoperationen }
SetFillStyle(UserFill, Green);  {setzt wieder Muster X}
```

Grenzen — vor dem Aufruf von *SetFillPattern* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *Bar, Bar3D, FillPoly, GetFillPattern, GetFillSettings, PieSlice, SetFillStyle.*

Beispiel

```
program SetFillPatternDemo;
uses Graph;
var
  GraphMode, GraphDriver: Integer;
const
  Gray50 : FillPatternType = ($AA, $55, $AA, $55,
                              $AA, $55, $AA, $55);
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
  SetFillPattern(Gray50, Green);
  Bar3D (40,20,100,160,20, TopOn);  { ein grüner Balken
                                      mit Schachbrettmuster }

  Readln;
  CloseGraph;
end.
```

SetFillStyle-Prozedur — Graph

Funktion — setzt ein vordefiniertes Muster zur Flächenfüllung.

Deklaration

```
SetFillStyle(Pattern: Word; Color: Word)
```

Erläuterung — *Color* ist ein Ausdruck des Typs *Word* und legt die Farbe fest, mit der nachfolgende Flächenfüllungen ausgeführt werden sollen. *Pattern* ist ebenfalls ein Ausdruck des Typs *Word* und legt eines der folgenden Muster fest:

```
const
  EmptyFill      = 0;  { Füllen mit Hintergrundfarbe }
  SolidFill      = 1;  { komplettes Ausfüllen }
  LineFill       = 2;  { --- }
  LtSlashFill    = 3;  { /// }
  SlashFill      = 4;  { /// mit dicken Linien }
  BkSlashFill    = 5;  { \\\ mit dicken Linien }
  LtBkSlashFill  = 6;  { \\\ }
  HatchFill      = 7;  { leicht schraffiert }
  XHatchFill     = 8;  { stark schraffiert, kreuzend }
  InterleaveFill = 9;  { abwechselnde Linien }
  WideDotFill    = 10; { Punkte, weit auseinander }
  CloseDotFill   = 11; { Punkte, dicht }
  UserFill       = 12; { benutzerdefiniert }
```

Hinweis: Ein Aufruf mit dem Wert *UserFill* setzt das Füll-Muster, das mit dem jeweils letzten Aufruf von *SetFillPattern* definiert wurde.

Grenzen — vor dem Aufruf von *SetFillStyle* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *Bar, Bar3D, FillPoly, GetFillPattern, GetFillSettings, PieSlice, SetFillPattern.*

Beispiel

```
program SetFillStyleDemo;
uses Crt, Graph;

var
  GraphMode, GraphDriver: Integer;
  x: Integer; ch : Char;

begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
  for x:= EmptyFill to CloseDotFill do
  begin
    SetFillStyle(x, x);
    Bar3D (40,20,160,160,20, TopOn);
    ch := ReadKey;  { Warten auf einen Tastendruck }
  end;
```

```
  Readln;
  CloseGraph;
end.
```

SetFTime-Prozedur — Dos

Funktion — setzt Datum und Uhrzeit der letzten Veränderung einer Datei direkt.

Deklaration — `SetFTime(var f; Time: LongInt)`

Erläuterung — *f* steht für eine Datei-Variable beliebigen Typs, die entsprechende Datei muß offen sein. Der *LongInt*-Ausdruck *Time* enthält Datum und Uhrzeit in einem speziellen gepackten Format (siehe Kapitel 12 »Standard-Units« des *Programmierhandbuchs*), das sich am einfachsten durch die Anwendung der Prozedur *PackTime* auf einen Record des Typs *DateTime* erzeugen läßt.

Fehler bei der Ausführung von *SetFTime* setzen die globale Variable *DosError*, der einzig mögliche Code ist hier 6 (ungültiger Handle/Datei nicht offen).

Grenzen — die durch *f* bezeichnete Datei muß offen sein.

Querverweis — *GetFTime*, *PackTime*, *UnpackTime*

SetGraphBufSize-Prozedur — Dos

Funktion — legt die Größe des Puffers für Flächenfüllungen und Polygone fest.

Deklaration — `SetGraphBufSize(BufSize: Word)`

Erläuterung — Die Routinen zur Flächenfüllung und zum Zeichnen von Polygonen verwenden einen internen Puffer, der von *InitGraph* (via *GraphGetMem*) beim Start des Grafikpakets auf dem Heap belegt und von *CloseGraph* (via *GraphFreeMem*) wieder freigegeben wird.

Standardmäßig umfaßt dieser Puffer 4096 Bytes – über einen Aufruf von *SetGraphBufSize*, der *vor* dem Start des Grafikpakets stattfinden muß, läßt er sich beliebig verkleinern (auf minimal 0 Bytes) oder vergrößern (auf maximal 64 KBytes). Eine Vergrößerung dürfte nur für extrem große Polygone bzw. extrem komplexe Flächenfüllungen notwendig sein: *FillPoly* kann bereits in 4096 Bytes rund 630 Eckpunkte speichern.

Grenzen — nach der Initialisierung des Grafikpakets via *InitGraph* werden Aufrufe von *SetGraphBufSize* vollständig ignoriert.

Querverweis — *FloodFill*, *FillPoly*, *InitGraph*.

SetGraphMode-Prozedur — Graph

Funktion — schaltet in einen Grafikmodus um und löscht den Bildschirm.

Deklaration

```
SetGraphMode(Mode: Integer)
```

Erläuterung — *Mode* ist ein Ausdruck des Typs Integer, der zulässige Wertebereich hängt vom verwendeten Grafiktreiber ab.

SetGraphMode aktiviert den durch *Mode* angegebenen Grafikmodus und setzt sämtliche Parameter des Grafikpakets (Zeichenfarbe und -fenster, Grafik-Cursor usw.) auf die vorgegebenen Standardwerte zurück.

Sinnvoll ist die Verwendung dieser Prozedur dann, wenn entweder ein anderer als der von *InitGraph* vorgegebene Grafikmodus gesetzt oder zwischen »Text« und »Grafik« umgeschaltet werden soll.

Nach dem Versuch, mit *SetGraphMode* einen für den momentan benutzten Treiber nicht definierten Modus zu setzen, liefert die Funktion *GraphResult* das Ergebnis -10.

In der Unit *Graph* sind die folgenden Konstanten für Grafikmodi definiert:

Name	Wert	Beschreibung
Grafiktreiber CGA		
CGAC0	0	320*200, Palette C0, 4 Farben, 1 Seite
CGAC1	1	320*200, Palette C1, 4 Farben, 1 Seite
CGAC2	2	320*200, Palette C2, 4 Farben, 1 Seite
CGAC3	3	320*200, Palette C3, 4 Farben, 1 Seite
CGAHI	4	640*200, 2 Farben, 1 Seite
Grafiktreiber MCGA		
MCGAC0	0	320*200, Palette C0, 4 Farben, 1 Seite
MCGAC1	1	320*200, Palette C1, 4 Farben, 1 Seite
MCGAC2	2	320*200, Palette C2, 4 Farben, 1 Seite
MCGAC3	3	320*200, Palette C3, 4 Farben, 1 Seite
MCGAMed	4	640*200, 2 Farben, 1 Seite
MCGAHi	5	640*480, 2 Farben, 1 Seite
Grafiktreiber EGAVGA, EGA mit 256 KByte RAM		
EGALo	0	640*200, 16 Farben, 4 Seiten
EGAHi	1	640*350, 16 Farben, 2 Seiten
EGAMonoHi	3	640*350, 2 Farben, 2 Seiten
Grafiktreiber EGAVGA, EGA mit 64 KByte RAM		
EGA64Lo	0	640*200, 16 Farben, 1 Seite
EGA64Hi	1	640*350, 4 Farben, 1 Seite
EGAMonoHi	3	640*350, 2 Farben, 1 Seite

Name	Wert	Beschreibung
Grafiktreiber EGAVGA, VGA		
VGALo	0	640*200, 16 Farben, 2 Seiten
VGAMed	1	640*350, 16 Farben, 2 Seiten
VGAHi	2	640*480, 16 Farben, 1 Seite
Grafiktreiber HERC		
HercMonoHi	0	720*348, 2 Farben, 2 Seiten
Grafiktreiber PC3270		
PC3270Hi	0	720*350, 2 Farben, 1 Seite
Grafiktreiber ATT400		
ATT400C0	0	320*200, Palette C0, 4 Farben, 1 Seite
ATT400C1	1	320*200, Palette C1, 4 Farben, 1 Seite
ATT400C3	2	320*200, Palette C2, 4 Farben, 1 Seite
ATT400C3	3	320*200, Palette C3, 4 Farben, 1 Seite
ATT400Med	4	640*200, 2 Farben, 1 Seite
ATT400Hi	5	640*400, 2 Farben, 1 Seite
Grafiktreiber IBM8514		
IBM8514Lo	0	640*480, 256 Farben, 1 Seite
IBM8514Hi	1	1024*768, 16 Farben, 1 Seite

Mit *GetModeRange* und *GetMaxMode* läßt sich ermitteln, welche Grafikmodi für einen bestimmten Treiber zulässig sind.

Grenzen — vor dem Aufruf von *SetGraphMode* muß das Grafikpaket mit *InitGraph* initialisiert worden sein.

Querverweis — *ClearDevice*, *DetectGraph*, *GetGraphMode*, *InitGraph*, *RestoreCrtMode*, *GraphResult*, *GetMaxMode*, *GetModeRange*.

Beispiel

```
....
  SetGraphMode(EGAHi);  { Grafik }
  ....
  RestoreCrtMode;  { zurück auf »Text« }
  ....
  SetGraphMode(EGAMonoHi);  { anderer Grafikmodus }
  ....
```

SetIntVec-Prozedur — Dos

Funktion — setzt einen Interrupt-Vektor des Systems auf eine bestimmte Adresse.

Deklaration

```
SetIntVec(IntNo: Byte; Vector: Pointer)
```

Erläuterung — *IntNo* ist ein Ausdruck des Typs *Byte* und gibt die Nummer des Vektors an, der auf die durch *Vector* festgelegte Adresse gesetzt werden soll.

Vector läßt sich entweder über die Funktion *Addr,* über den Adress-Operator @ oder über *Ptr* erzeugen.

Der Hauptzweck der Routinen *GetIntVec* und *SetIntVec* besteht in einer *zeitweiligen* »Umleitung« eines Interrupt-Vektors. Das im »Beispiel« gezeigte Programm leitet den durch Shift PrtSc ausgelösten Interrupt auf eine eigene Routine um und könnte auch verwendet werden, um einen Drucker für Bildschirmkopien anzupassen.

Vorsicht: Der weitaus größte Teil der Interrupt-Vektoren wird nach dem Ende eines Programms **nicht** automatisch neu gesetzt. Wenn das Programm vor seinem Ende nicht dafür sorgt, daß diese Vektoren wieder ihre Originalwerte bekommen, führt der nächste entsprechende Interrupt »ins Leere« – nämlich dahin, wo die entsprechende Behandlungsroutine einmal stand, als das Programm noch im Speicher war. Die mildeste Konsequenz ist ein völliger Absturz des Systems.

Kapitel 14 »Interne Details« des *Programmierhandbuchs* setzt sich eingehender mit **interrupt**-Prozeduren auseinander.

Querverweis *GetIntVec*

Beispiel

```
program SetIntVecDemo;
uses Dos;
var  PrintScreenVec: Pointer;

procedure PrintScreenNeu;  { für dieses Beispiel }
interrupt;                 {   parameterlos      }
begin
{ Hier könnte auch eine Bildspeicher-Auslese,
  Umcodierung und anschließende Druckerausgabe stehen }
end;

begin  { Hauptprogramm }
  GetIntVec(5,PrintScreenVec);   { Original speichern }
  SetIntVec(5, @PrintScreenNeu); { eigene Routine }
  Writeln('Print Screen-Interrupt umgeleitet.');
  Writeln('Probieren Sie es doch einmal aus');

  Readln;
  SetIntVec(5, PrintScreenVec);  { Original zurück! }
end.
```

SetLineStyle-Prozedur — Graph

Funktion setzt die Linienart und -dicke für folgende Zeichenaktionen.

Deklaration `SetLineStyle(LineStyle , Pattern, Thickness: Word)`

Erläuterung alle drei Parameter dieser Prozedur sind Ausdrücke des Typs Word. *LineStyle* legt die Art der Linie fest, die in der Unit *Graph* mit folgenden Möglichkeiten definiert ist:

```
const
  { Möglichkeiten für den Parameter LineStyle }
  SolidLn   = 0;   { durchgezogene Linie }
  DottedLn  = 1;   { gepunktet }
  CenterLn  = 2;   { Punkt - Strich - Punkt }
  DashedLn  = 3;   { gestrichelt }
  UserBitLn = 4;   { benutzerdefiniert }
```

Der Parameter *Pattern* hat nur dann eine Wirkung, wenn *LineStyle* den Wert *UserBitLn* hat: in diesem Fall legen die einzelnen Bits das Punktmuster der Linie fest, wobei das höchstwertige Bit dem jeweils zuerst gezeichneten Punkt einer Linie entspricht. Der Parameter *Thickness* legt die Dicke der Linie fest. Hier sind nur zwei Konstanten in der Unit *Graph* definiert, nämlich

```
const
  NormWidth  = 1;  { normale Dicke }
  ThickWidth = 3;  { dicke Dicke   }         { Aua...}
```

Die mit SetLineStyle gesetzte Linienart gilt für *alle* folgenden Linien - also nicht nur für *Line* und *LineTo*, sondern auch für die Prozeduren *Arc*, *Circle*, *DrawPoly*, *Rectangle* usw.

Grenzen

vor dem Aufruf von *SetLineStyle* muß ein Grafikmodus gesetzt worden sein.

Querverweis

DrawPoly, GetLineSettings, Line, LineRel, LineTo, GraphResult, SetWriteMode.

Beispiel

```
program SetLineStyleDemo;
uses Graph;
var
  GraphMode, GraphDriver: Integer;
  x,y: Integer;
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
  { eine benutzerdefinierte Linie }
  SetLineStyle(UserBitLn,$F555,NormWidth);
  Line(0,10,100,10);
  for x:= SolidLn to DashedLn do begin
    SetColor(x+1); SetLineStyle(x,0,NormWidth);
    for y := 0 to 40 do
      Line(0,x*40+y+20,320,x*40+y+20);
   end;
  Readln;
  CloseGraph;
end.
```

SetPalette-Prozedur Graph

Funktion ändert einen Eintrag der momentan gesetzten Farb-Palette.

Deklaration
```
SetPalette(ColorNum: Word; ColorShortInt);
```

Erläuterung *SetPalette* setzt den durch *ColorNum* bezeichneten Eintrag der Palette auf die Farbe *Color*: ein Aufruf wie *SetPalette*(0,*LightBlue*) legt also fest, daß alle mit der »Farbe 0« gezeichneten Objekte bis auf weiteres in Hellblau zu erscheinen haben. Mögliche Werte für *ColorNum* und *Color* liegen im Bereich von 0..15, der tatsächlich erlaubte Wertebereich ist vom verwendeten Grafiktreiber und -modus abhängig. Beim Aufruf mit unzulässigen Werten läßt *SetPalette* die momentan gesetzte Farb-Palette unverändert; ein folgender Aufruf von *GraphResult* liefert das Ergebnis -11 (*grError*).

Die folgenden Konstanten und Datentypen sind in der Unit *Graph* für die Arbeit mit Farb-Paletten definiert:

```
const
  Black        = 0;   { Schwarz }
  Blue         = 1;   { Blau }
  Green        = 2;   { Grün }
  Cyan         = 3;   { Türkis }
  Red          = 4;   { Rot }
  Magenta      = 5;   { Fuchsinrot }
  Brown        = 6;   { Braun }
  LightGray    = 7;   { Hellgrau }
  DarkGray     = 8;   { Dunkelgrau }
  LightBlue    = 9;   { Hellblau }
  LightGreen   = 10;  { Hellgrün }
  LightCyan    = 11;  { helles Türkis }
  LightRed     = 12;  { Hellrot }
  LightMagenta = 13;  { helles Fuchsinrot }
  Yellow       = 14;  { Gelb }
  White        = 15;  { Weiß }

MaxColors = 15;  { maximal 16 Paletten-Einträge }

type
  PaletteType = record
      Size   : Byte;
      Colors : array[0..MaxColors] of ShortInt;
    end;
```

Veränderungen innerhalb der Farb-Palette werden sofort auf dem Bildschirm sichtbar. Das als »Beispiel« gezeigte Programm zeichnet einige Balken in verschiedenen Farben und ändert danach die entsprechenden Einträge der Farb-Palette.

Grenzen vor dem Aufruf von *SetPalette* muß ein Grafikmodus gesetzt worden sein. Für die IBM-8514-Karte sowie den VGA-Modus mit 256 Farben muß die Prozedur *SetRGBPalette* anstelle von *SetPalette* verwendet werden.

Querverweis *GetBkcolor, GetColor, GetPalette, SetAllPalette, SetBkColor, SetColor, SetRGBPalette.*

Beispiel

```
program  SetPaletteDemo;
usesCrt,  Graph;
var
  GraphMode, GraphDriver: Integer;
  Palette: PaletteType;
  x,y : Integer;
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');

  GetPalette(Palette);
  for x:= 1 to 9 do begin
    SetFillStyle(x, x mod 4 + 1);
    Bar(x * 60, 20, (x+1) * 60, 200);
  end;
  repeat
    SetPalette(Random(Palette.Size),
               Random(Palette.Size));
    Delay(300);
  until KeyPressed;
  Readln;
  CloseGraph;
end.
```

SetRGBPalette-Prozedur Graph

Funktion ändert einen Eintrag der momentan gesetzten Farb-Palette für den IBM-Adapter 8514 und für VGA-Karten im Modus mit 256 Farben.

Deklaration

```
SetRGBPalette(ColorNum, RedValue , GreenValue,
              BlueValue: Integer);
```

Erläuterung der Integer-Ausdruck *ColorNum* hat einen zulässigen Bereich von 0..255 und legt die Nummer des Farb-Eintrags fest, dessen Wert sich aus den Komponenten *RedValue, GreenValue* und *BlueValue* zusammensetzt.

Der IBM-Adapter 8514 ist in seiner Ansteuerung bereits für zukünftige Erweiterungen ausgelegt – deshalb fallen die Wertebereiche der drei Komponenten etwas merkwürdig aus: interpretiert werden jeweils die oberen sechs Bits des nieder-

wertigen Bytes – mögliche Werte für *RedValue*, *GreenValue* und *BlueValue* sind also 0, 4, 8 .. 256.

Hinweis: Der Treiber IBM8514.BGI initialisiert die ersten 16 Paletten-Einträge des IBM-8514-Adapters mit Werten, die den Standardfarben einer EGA- bzw. VGA-Karte entsprechen und sorgt so dafür, daß »normale« Programme auch mit diesem Adapter lauffähig sind. (Dabei handelt es sich um eine reine Initialisierung – auch diese 16 Einträge lassen sich mit *SetRGB-Palette* beliebig verändern).

Im Prinzip läßt sich *SetRGBPalette* auch auf VGA-Karten anwenden: der mögliche Wertebereich für *ColorNum* reduziert sich hier auf 0..16; Interpretation und Wertebereiche der drei Komponenten *RedValue*, *GreenValue* und *BlueValue* sind dieselben wie beim IBM-8514-Adapter.

Grenzen

vor dem Aufruf von *SetRGBPalette* muß ein Grafikmodus gesetzt worden sein. Diese Prozedur ist nur auf Systemen anwendbar, die mit einem IBM-8514-Adapter oder einer VGA-Karte ausgerüstet sind.

Querverweis

GetBkColor, *GetColor*, *SetBkcolor*, *SetColor*, *SetPalette*.

Beispiel

```
program SetRGBPaletteDemo;
uses Graph;
type
  RGBRec = record
    RedVal, GreenVal, BlueVal: Integer;
  end;

const
 EGAColors: array[0..MaxColors] of RGBRec =
  {  Komponenten                          Name | Farbe }
  ( (RedVal:$00;GreenVal:$00;BlueVal:$00),{ Black   0  }
    (RedVal:$00;GreenVal:$00;BlueVal:$FC),{ Blue    1  }
    (RedVal:$24;GreenVal:$FC;BlueVal:$24),{ Green   2  }
    (RedVal:$00;GreenVal:$FC;BlueVal:$FC),{ Cyan    3  }
    (RedVal:$FC;GreenVal:$14;BlueVal:$14),{ Red     4  }
    (RedVal:$B0;GreenVal:$00;BlueVal:$FC),{ Magenta 5  }
    (RedVal:$70;GreenVal:$48;BlueVal:$00),{ Brown   20 }
    (RedVal:$C4;GreenVal:$C4;BlueVal:$C4),{ White   7  }
    (RedVal:$34;GreenVal:$34;BlueVal:$34),{ Gray    56 }
    (RedVal:$00;GreenVal:$00;BlueVal:$70),{ L-Blue  57 }
    (RedVal:$00;GreenVal:$70;BlueVal:$00),{ L-Green 58 }
    (RedVal:$00;GreenVal:$70;BlueVal:$70),{ L-Cyan  59 }
    (RedVal:$70;GreenVal:$00;BlueVal:$00),{ L-Red   60 }
    (RedVal:$70;GreenVal:$00;BlueVal:$70),{ L-Mag.  61 }
    (RedVal:$FC;GreenVal:$FC;BlueVal:$24),{ Yellow  62 }
    (RedVal:$FC;GreenVal:$5C;BlueVal:$FC) { Brown   63 }
  );
```

```
var
  GraphDriver, GraphMode: Integer;
  x : Integer;

begin
  GraphDriver := IBM8514;   { *Ausschalten* der }
  GraphMode := IBM8514Hi;   { automatischen Erkennung }
  InitGraph(GraphDriver, GraphMode, '\TP');

  if GraphResult < 0 then
  begin
    Writeln('Weder 8514 noch VGA-Karte installiert!');
    Halt(1);
  end;

{ Textausgabe in verschiedenen Farben }
  for x := 1 to MaxColors do
  begin
    SetColor(x);
    OutTextXY(20, (x+1) * 10, 'Irgendein Text');
  end;

{ die Farben etwas durcheinanderbringen }
  for x:= 1 to 100 do
    SetRGBPalette(x,x*4,x*4,x*4);
  Readln;

  { Wiederherstellen der ersten 16 Farben }
  for x:= 0 to MaxColors do
    with EGAColors[x] do
      SetRGBPalette(x, RedVal, GreenVal, BlueVal);

  Readln;
  CloseGraph;
end.
```

SetTextBuf-Prozedur

Funktion — weist einer Textdatei-Variablen einen Puffer zu.

Deklaration — `SetTextBuf(var f: Text; var buf [; size: Word] )`

Erläuterung — *f* ist eine Textdatei-Variable, *buf* ist eine Variable beliebigen Typs, *size* ein optionaler *Word*-Ausdruck.

Jede Textdatei-Variable verfügt über einen eigenen Puffer von 128 Bytes Größe, der standardmäßig von *Read* und *Write* benutzt wird. Für die meisten Anwendungen sind Puffer dieser Größe ausreichend; falls Sie allerdings ganze Textdateien zei-

lenweise kopieren wollen oder andere ähnlich aufwendige Datei-Operationen vorhaben, sollte der Puffer auf einige KByte vergrößert werden. Das Programm wird durch größere Dateipuffer schneller.

SetTextBuf verändert die Dateivariable *f* so, daß die als *buf* angegebene Variable zur Zwischenspeicherung benutzt wird. Wenn der Parameter *size* angegeben ist, wird der entsprechende Teil von *buf* benutzt; ansonsten setzt der Compiler anstelle von *size* automatisch die Größe der Variablen *buf* ein (also *SizeOf*(*buf*)).

Die Datei-Variable benutzt einen mit *SetTextBuf* gesetzten Puffer solange, bis sie mit *Assign* erneut zugeordnet (und damit initialisiert) wird.

Vorsicht: Turbo Pascal kann nicht bestimmen, ob eine als Puffer zugeordnete Variable diesen Zweck auch während der gesamten Laufzeit des Programms erfüllen kann. Einer der häufigsten Fehler in diesem Zusammenhang ist die Deklaration der Puffervariablen innerhalb einer lokalen Routine, also:

```
procedure Neuer_Puffer (var f: Text);
var
  Puffer: array[1..2048] of Char;    { FALSCH ! }
begin
  SetTextBuf(f, Puffer);
   ....
end;  { ab hier ist "Puffer" undefiniert }
```

Grenzen

SetTextBuf sollte nie auf eine offene Datei angewendet werden – da die Puffer-Variable einfach ausgetauscht wird, gehen sonst Daten verloren (ein Aufruf *direkt* nach einem *Reset*, *Rewrite* oder *Append* ist allerdings möglich). *f* muß eine Textdatei bezeichnen.

Beispiel

```
program SetTextBufDemo;
var
  f : Text;
  ch : Char;
  buf: array[1..10240] of Char;  { 10 KByte }
begin
  { Dateiname wird als 1. Kommandozeilen-
    Parameter angenommen }
  Assign(f, ParamStr(1));
  SetTextBuf(f, buf); { Zuordnung des größeren Puffers }
  Reset(f);

  { "Und plötzlich ging alles rasend schnell. Es war
     einfach schrecklich", sagte Erna W. (53), die
     einzige Zeugin. Reporter Esser ...}
  while not Eof(f) do
```

```
  begin
    Read(f, ch);
    Write(ch);
  end;
  Writeln;  Close(f);
end.
```

SetTextJustify-Prozedur — Graph

Funktion — legt die Ausrichtung von Textausgaben durch *OutText* und *OutTextXY* fest.

Deklaration

```
SetTextJustify(Horiz , Vert: Word)
```

Erläuterung — die durch *SetTextJustify* gesetzten Parameter bestimmen, in welchem Verhältnis zur Position des Grafik-Cursors Textausgaben über *OutText* bzw. *OutTextXY* geschehen. Eine Befehlsfolge wie

```
SetTextJustify(CenterText, CenterText);
OutTextXY(100, 100, 'ABC');
```

schreibt die Zeichen *ABC* so auf den Bildschirm, daß der Punkt (100,100) im Zentrum des Buchstabens *B* liegt.

Für die Ausrichtung von Texten definiert *Graph* folgende Konstanten:

```
const
  { horizontal }
LeftText   = 0;   { Text beginnt ab GC-Position }
CenterText = 1;   { Text mittig zum GC }
RightText  = 2;   { Text endet mit GC-Position }

  { vertikal }
BottomText = 0;   { Text oberhalb GC-Position }
{ CenterText = 1;   mittig zum GC, siehe oben }
TopText    = 2;   { Text unterhalb GC-Position }
```

Grenzen — vor dem Aufruf von *SetTextJustify* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *GetTextSettings, OutText, OutTextXY, SetTextStyle, SetUserCharSize, TextHeight, TextWidth.*

Beispiel

```
program SetTextJustifyDemo;
uses Graph;
var
  GraphMode, GraphDriver: Integer;
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
```

```
  SetTextJustify(CenterText, CenterText);
  LineTo(GetMaxX, GetMaxY);
  LineTo(GetMaxX,0); LineTo(0,GetMaxY);
  OutTextXY(Succ(GetMaxX) div 2, Succ(GetMaxY) div 2,
            'In der Mitte');
  Readln;
  CloseGraph;
end.
```

SetTextStyle-Prozedur — Graph

Funktion — legt den Zeichensatz, die Rotation und die Größe folgender Textausgaben fest.

Deklaration

```
SetTextStyle(Font , Direction: Word; CharSize: Word);
```

Erläuterung — alle drei Parameter von *SetTextStyle* sind Ausdrücke des Typs *Word*: *Font* legt den Zeichensatz fest (und ist gegenwärtig auf Werte im Bereich von 0..4 beschränkt, solange nicht eigene Zeichensätze via *InstallUserFont* mit in das Programm aufgenommen werden), *CharSize* bestimmt den Vergrößerungsfaktor der durch *Font* festgelegten Zeichen. (Standardvorgabe für *DefaultFont*: 1, für die Vektor-Zeichensätze: 4). Über die Prozedur *SetUserCharSize* lassen sich voneinander unabhängige Vergrößerungsfaktoren in X- und Y-Richtung setzen.

Die Rotation von Textausgaben wird über den Parameter *Direction* bestimmt: der Wert *HorizDir* erzeugt Ausgaben von links nach rechts; der Wert *VertDir* erzeugt Ausgaben von unten nach oben.

Das Grafikpaket verwendet einen Standard-Zeichensatz, dessen Zeichen in einem 8 * 8-Raster bitweise definiert sind. Vergrößerungen dieser Zeichen vergröbern die Auflösung immer weiter (siehe Kapitel 12 »Standard-Units« des *Programmierhandbuchs*). Die restlichen Zeichensätze arbeiten deshalb mit *Vektoren*, d.h. jedes Zeichen ist als Abfolge relativer Bewegungen gespeichert. Der Sinn dieses erhöhten Rechenaufwandes liegt hauptsächlich darin, daß Vergrößerungen dieser Zeichensätze nicht automatisch eine Vergröberung der einzelnen Zeichen nach sich ziehen.

Vektor-Zeichensätze werden – solange sie nicht über *RegisterBGIFont* direkt als Teil des Programms »registriert« sind – nach Bedarf von der Diskette geladen und im Heap-Bereich gespeichert. Ein Aufruf von *SetTextStyle* kann daher eine ganze Reihe von verschiedenen Fehlerbedingungen (d.h. Funktionsergebnisse von *GraphResult*) erzeugen:

-8 Zeichensatz-Datei nicht gefunden
-9 nicht genug Speicherplatz zum Laden des Zeichensatzes
-12 I/O-Fehler beim Lesen der .CHR-Datei

-13 Kopfinformation der .CHR-Datei ungültig
-14 unbekannte Zeichensatz-Nummer in der .CHR-Datei

Die Unit *Graph* definiert die folgenden Konstanten für Zeichensätze und Schreibrichtungen:

```
const
{ Set/GetTextStyle }
  DefaultFont   = 0;   { 8x8 Bit pixelweise definiert }

  TriplexFont   = 1;   { Vektor-Zeichensätze }
  SmallFont     = 2;
  SansSerifFont = 3;
  GothicFont    = 4;

  HorizDir   = 0;       { von links nach rechts }
  VertDir    = 1;       { von unten nach oben }
```

Grenzen vor dem Aufruf von *SetTextStyle* muß ein Grafikmodus gesetzt worden sein.

Querverweis *GetTextSettings, OutText, OutTextXY, RegisterBGIFont, SetTextJustify, SetttUserCharSize TextHeight, TextWidth.*

Beispiel

```
program SetTextStyleDemo;
uses Graph;
var
  GraphMode, GraphDriver: Integer;
  x, y: Integer;
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
  y:= 10;
  for x:= 2 to 8 do begin
    OutTextXY(0, y, 'Größe = ' + chr(x + 48));
    SetTextStyle(DefaultFont, HorizDir, x);
    Inc(y, TextHeight('a') + 1);
  end;
  Readln;
  CloseGraph;
end.
```

SetTime-Prozedur — Dos

Funktion setzt die Uhrzeit des Systems.

Deklaration `SetTime(Hour , Minute, Second, Sec100: Word)`

Erläuterung gültige Werte für *Hour* gehen von 0..23, für *Minute* und *Second* von 0..59 und für *Sec100* (hundertstel Sekunden) von 0..99.

Wenn auch nur einer der Parameter außerhalb dieser Grenzen liegt, bleibt die Uhrzeit des Systems unverändert.

Querverweis *GetDate, GetTime, SetDate*

SetUserCharSize-Prozedur — Graph

Funktion legt voneinander unabhängige Vergrößerungsfaktoren in X- und Y-Richtung für Grafik-Zeichensätze fest.

Deklaration
```
SetUserCharSize(MultX , DivX , MultY, DivY: Word)
```

Erläuterung *SetUserCharSize* erlaubt im Gegensatz zu *SetTextStyle* voneinander unabhängige Vergrößerungsfaktoren in X- und Y-Richtung: die Breite eines Zeichens wird zuerst mit *MultX* multipliziert und dann durch *DivX* (ganzzahlig) dividiert. Analoges gilt für die Zeichenhöhe und die Parameter *MultY* bzw. *DivY*.

Im folgenden Beispiel wird der Text *ABC* mit einer Breite von »8« und einer Höhe von »6.5« ausgegeben:

```
SetTextStyle(TriplexFont, HorizDir, 1);
SetUserCharSize(8,1,13,2);
OutTextXY(100,100, 'ABC');
```

Hinweis: Ab der Version 5.0 setzt *SetUserCharSize* die Zeichengröße mit sofortiger Wirkung – der in der Version 4.0 erforderliche Aufruf von *SetTextStyle* mit *CharSize* = 0 kann entfallen. (Um es genau zu nehmen: Er *muß* hinausgekürzt werden, weil *SetTextStyle* sonst wieder eine eigene Zeichengröße setzt).

Grenzen vor dem Aufruf von *SetUserCharSize* muß ein Grafikmodus gesetzt worden sein.

Querverweis *SetTextStyle, OutText, OutTextXY, TextHeight, TextWidth.*

Beispiel
```
program SetUserCharSizeDemo;
uses Graph;
var
  GraphDriver, GraphMode: Integer;
  x,y: Integer;

begin
  GraphDriver := Detect;   { automatische Erkennung }
  InitGraph(GraphDriver, GraphMode, '\TP');

  SetTextStyle(SansSerifFont, HorizDir, 4);
  SetTextJustify(CenterText, CenterText);
  for y:= MaxColors downto 1 do
  begin
    SetColor(y);
    for x:= 1 to 35 do
```

```
    begin
      SetUserCharSize(x,2,x,2);
      OutTextXY(GetMaxX div 2, 40, 'as');
    end;
  end;

  Readln;
  CloseGraph;
end.
```

SetVerify-Prozedur — Dos

Funktion — setzt das *Verify*-Flag von DOS.

Deklaration — `SetVerify(Verify: Boolean)`

Erläuterung — das Flag *Verify* legt fest, ob DOS Diskettensektoren nach dem Schreiben (durch eine reine Leseaktion ohne direkten Datenvergleich) noch einmal prüft oder nicht. Die Prozedur *SetVerify* setzt dieses Flag mit dem als *Verify* übergebenen Ausdruck:

True: DOS führt eine zusätzliche Überprüfung aus.

False: Diskettensektoren werden nach dem Schreiben nicht noch einmal zur Verifizierung der Prüfsumme gelesen.

Hinweis: Mit der Prozedur *GetVerify* läßt sich der momentane Wert dieses DOS-Flags ermitteln.

Querverweis — *GetVerify*

SetViewPort-Prozedur — Graph

Funktion — setzt ein Grafik-Zeichenfenster.

Deklaration — `SetViewPort(x1,y1,x2,y2: Integer; Clip: Boolean)`

Erläuterung — *(x1,y1)* legt die obere linke, *(x2,y2)* die untere rechte Ecke des Zeichenfensters fest. Diese vier *Integer*-Ausdrücke müssen folgende Bedingungen erfüllen:

```
0 <= x1 < x2 <= GetMaxX
0 <= y1 < y2 <= GetMaxY
```

Im Gegensatz zu fast allen anderen Positionierungsbefehlen arbeitet *SetViewPort* mit absoluten Koordinaten. Der Aufruf

```
SetViewport(0,0,100,100,True);
```

setzt ein Zeichenfenster von 100 Punkten Breite, das sich in der oberen linken Ecke des Bildschirms befindet – unabhängig davon, ob zuvor ein anderes Zeichenfenster gesetzt war oder nicht.

Der Parameter *Clip* ist ein Ausdruck des Typs *Boolean* und legt fest, ob Zeichenaktionen am Rand des Fensters abgeschnitten werden sollen oder nicht. Eine Befehlsfolge wie

```
SetViewPort(0,0,100,100,ClipOn); { Abschneiden aktiv }
LineTo(200,200);
```

zeichnet eine Linie vom (absoluten) Punkt (0,0) zum Punkt (100,100), der restliche Teil der Zeichenaktion (die Fortsetzung zum Punkt (200,200)) wird unterdrückt.

Hinweis: Der Grafik-Cursor selbst ist von einem »Clipping« nie betroffen, nur Zeichenaktionen werden gegebenenfalls unterdrückt. Im zuvor gezeigten Beispiel befindet sich der Grafik-Cursor nach dem *LineTo*-Befehl auf dem (absoluten) Punkt (200,200).

Die Unit *Graph* definiert die folgenden Konstanten für den Parameter *Clip*:

```
const
  ClipOn = True;   { "Clipping" aktiv }
  ClipOff = False; { nicht aktiv }
```

Jeder Aufruf von *SetViewPort* definiert ein neues Zeichenfenster und setzt den Grafik-Cursor automatisch auf den Ursprung dieses Fensters (relative Koordinaten (0,0)). Eine Veränderung des Bildschirminhalts an der Stelle des neuen Fensters nimmt *SetViewPort* nicht vor.

Die Prozeduren *InitGraph*, *GraphDefaults* und *SetGraphMode* setzen das Zeichenfenster bei jedem Aufruf auf den gesamten Bildschirm. Die Prozedur *GetViewSettings* ermittelt die Parameter des momentan gesetzten Zeichenfensters und liefert sie über einen Record des (in der Unit *Graph* definierten) Typs *ViewPortType* zurück:

```
type
  ViewPortType = record
    x1,y1,x2,y2  : Word;  { Fenstergrenzen }
    Clip:   Boolean;      { "Clipping" }
  end;
```

Grenzen — vor dem Aufruf von SetViewPort muß ein Grafikmodus gesetzt worden sein.

Querverweis — *ClearDevice*, *ClearViewPort*, *GetViewSettings*.

Beispiel

```
program SetViewPortDemo;
uses Crt, Graph;
var
  GraphMode, GraphDriver: Integer;
  x: Integer; ch: Char;
  x1,y1,x2,y2: Integer;
```

```
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');

  repeat      { zeichnet einen Hintergrund }
    PutPixel(Random(GetMaxX), Random(GetMaxY),
             Random(15));
  until KeyPressed;
  ch := ReadKey;

  x1 := GetMaxX - 90; y1 := GetMaxY - 90;
  SetColor(Red);
  repeat
    x2 := x1 + 80; y2 := y1 + 80;
    SetViewPort(x1,y1,x2,y2,ClipOn);
    ClearViewPort;
    Rectangle(40,20,80,75);
    Dec(x1, Round(GetMaxY/10));
    Dec(y1, Round(GetMaxY/20));
  until (x1 < 0) or (y1 < 0);

  Readln;
  CloseGraph;
end.
```

SetVisualPage-Prozedur — Graph

Funktion — wählt eine von mehreren Grafikseiten zur Anzeige aus.

Deklaration — `SetVisualPage(Page: Word)`

Erläuterung — mehrere Grafik-Seiten werden nur von EGA- und VGA- sowie von Hercules-Karten unterstützt. Mit der Prozedur *SetActivePage* läßt sich bei diesen Karten festlegen, auf welche Grafik-Seite *gezeichnet* wird, während *SetVisualPage* bestimmt, welche Seite *sichtbar* ist (beide Routinen sind unabhängig voneinander). Durch Zeichnen »im Hintergrund« und entsprechendes Umschalten der Seiten lassen sich bewegte Grafiken ohne die sonst unvermeidbaren Flimmereffekte realisieren.

Grenzen — vor dem Aufruf von *SetVisualPage* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *SetActivePage*.

Beispiel

```
....
SetVisualPage(0);   { für den Benutzer }
SetActivePage(1);
  ....       { Zeichenaktionen im Hintergrund }
SetVisualPage(1);  { wird jetzt gezeigt }
```

```
SetActivePage(0);
  ....       { jetzt wird die Seite 0
               "unsichtbar" bearbeitet }
```

SetWriteMode-Prozedur — Graph

Funktion — legt fest, ob Linien-Zeichenoperationen den vorherigen Bildinhalt überschreiben oder als Verknüpfung ausgeführt werden.

Deklaration

```
SetWriteMode(WriteMode: Integer)
```

Erläuterung — *SetWriteMode* legt fest, auf welche Weise die Routinen *DrawPoly*, *Line*, *LineRel*, *LineTo* und *Rectangle* ihre Linien zeichnen: mit direktem Überschreiben (Modus *CopyPut*) oder durch eine XOR-Operation mit dem vorherigen Bildinhalt (Modus *XORPut*). Im zweiten Fall lassen sich die gezeichneten Linien durch einen weiteren Zeichenbefehl im selben Modus ohne Zerstörung des Hintergrundes wieder vom Bildschirm entfernen.

Die beiden möglichen Werte für den Ausdruck *WriteMode* sind in der Unit *Graph* als Konstanten definiert:

```
const
  CopyPut = 0;   { MOV-Befehl = Überschreiben }
  XORPut  = 1;   { XOR-Operation mit dem Bildinhalt }
```

Grenzen — vor dem Aufruf von *SetWriteMode* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *GetLineSettings*, *Line*, *Rectangle*, *SetLineStyle*.

Beispiel

```
program SetWriteModeDemo;
uses Graph;
var
  GraphDriver, GraphMode: Integer;
  PPoints: array[0..29,0..1] of Integer;

  x: Integer;

procedure DrawLines;  { Streifen über den Bildschirm }
begin
  x := 0;
  while (x < GetMaxY) do
  begin
    Line(0, x, GetMaxX, x);
    Inc(x, 10);
   end;
end;

begin
  GraphDriver := Detect;   { automatische Erkennung }
  InitGraph(GraphDriver, GraphMode, '\TP');
```

```
  SetColor(Blue);    { ein ausgefüllter Kreis }
  SetFillStyle(SlashFill, Blue);
  Circle(GetMaxX div 2, GetMaxY div 2, GetMaxY div 3);
  FloodFill(GetMaxX div 2, GetMaxY div 2, Blue);

  Randomize;
  for x:= 0 to 29 do  { ein recht komplexes Polygon }
  begin
    PPoints[x,0] := Random(GetMaxX);
    PPoints[x,1] := Random(GetMaxY);
  end;

  SetFillStyle(LtSlashFill, LightRed);
  SetColor(Green);      { Zeichnen des Polygons }
  FillPoly(30,PPoints);

  Readln;      { Damit wäre der Hintergrund fertig }

  SetLineStyle(SolidLn, 0, ThickWidth);
  SetColor(White);
  SetWriteMode(XorPut);   { XOR-Operationen }

DrawLines;  { nun wird er ziemlich gründlich ruiniert }
Readln;
DrawLines;  { und dann wiederhergestellt }

  Readln;
  CloseGraph;
end.
```

Sin-Funktion

Funktion	liefert den Sinus des Arguments zurück.
Deklaration	`Sin(x: Real)`
Ergebnistyp	*Real*
Erläuterung	*x* ist ein Real-Ausdruck und stellt einen Winkel in der Einheit *Rad* (0..2 Pi) dar. Das Ergebnis der Funktion ist der Sinus von *x*.
Querverweis	*ArcTan, Cos*

Beispiel	`var x : Real;`
	`begin`
	`  x := Sin(Pi);  { 0 }`
	`end.`

SizeOf-Funktion

Funktion liefert die Anzahl von Bytes zurück, die das Argument an Speicherplatz belegt.

Deklaration `SizeOf(x)`

Ergebnistyp *Word*

Erläuterung *x* ist entweder ein beliebiger Variablenbezug oder ein Typ-Bezeichner. Das Ergebnis der Funktion ist der Platzbedarf von *x* in Bytes.

Wird diese Funktion auf eine Instanz eines Objekttyps mit einer virtuellen Methodentabelle (VMT) angewendet, dann liefert *SizeOf* die Größe, die in der VMT gespeichert ist. Damit erhält man also für Objekttypen, die eine VMT haben, die *tatsächliche* Größe der Instanz und nicht die *deklarierte* Größe.

Zur Angabe von Bereichsgrößen als Parameter von *FillChar, Move, GetMem* usw. sollte *SizeOf* anstelle von Konstanten verwendet werden:

```
FillChar(s, SizeOf(s), 0);
GetMem(p, SizeOf(Record_Typ));
```

Querverweis *GetMem*, *FillChar*, *Move*

Beispiel

```
type
  Eintrag = record
     Name   : string[30];
     Telefon: string[14];
    end;
var
  p : ^Eintrag;
begin
  GetMem(p, SizeOf(Eintrag));
  ...
end.
```

Sound-Prozedur Crt

Funktion aktiviert den eingebauten Lautsprecher.

Deklaration `Sound(Hz: Word);`

Erläuterung *Hz* ist ein Ausdruck des Typs *Word* und gibt die Frequenz des zu erzeugenden Tons in der Einheit Hertz (Schwingungen pro Sekunde) an. Ein mit *Sound* gestarteter Ton wird solange erzeugt, bis der Lautsprecher durch einen Aufruf von *NoSound* wieder abgeschaltet wird.

Querverweis *NoSound*, *Delay*.

Beispiel

```
uses Crt;
begin
  Sound(220); { Piepton an }
  Delay(200); { hier können beliebige Aktionen stehen }
  NoSound;  { Ruhe! }
end.
```

SPtr-Funktion

Funktion	liefert den momentanen Wert des Stackzeigers (SP-Register) zurück.
Deklaration	`SPtr`
Ergebnistyp	*Word*
Erläuterung	das Ergebnis von *SPtr* entspricht dem *Offset*-Anteil der Spitze des Stacks, d.h. dem Stackzeiger innerhalb des Stack-Segments.
Querverweis	*SSeg*

Sqr-Funktion

Funktion	liefert das Quadrat des Arguments zurück.
Deklaration	`Sqr(x)`
Ergebnistyp	derselbe Typ wie der des Arguments.
Erläuterung	*x* ist entweder ein Integer- oder ein Real-Ausdruck. Das Ergebnis der Funktion hat denselben Typ wie *x* und entspricht der Berechnung *x* * *x*.

Sqrt-Funktion

Funktion	liefert die Quadratwurzel (square root) des Arguments zurück.
Deklaration	`Sqrt(x: Real)`
Ergebnistyp	*Real*
Erläuterung	x ist ein Ausdruck des Typs *Real*. Das Ergebnis der Funktion ist die Quadratwurzel von *x*.
Grenzen	negative Werte von *x* sind mathematisch nicht definiert und erzeugen einen Laufzeitfehler.

SSeg-Funktion

Funktion	liefert die Adresse des Stack-Segments zurück.
Deklaration	`SSeg`

Ergebnistyp *Word*

Erläuterung das Ergebnis der Funktion ist der Inhalt des Prozessor-Registers SS.

Querverweis *CSeg, DSeg, SPtr*

Str-Prozedur

Funktion konvertiert einen numerischen Wert in einen String.

Deklaration `Str(x [ : width [ : decimals ] ], var s: string)`

Erläuterung *x* stellt den zu konvertierenden Wert dar und ist ein Integer- oder Real-Ausdruck. *Str* bildet einen String, der dieselbe Folge von Zeichen enthält, wie sie *Write* (ohne Angaben von *Breite* und *MinStellen*, siehe »Write-Prozedur) ausgeben würde – tatsächlich verwenden beide dieselbe Routine der Laufzeitbibliothek zur Erzeugung numerischer Zeichenfolgen. Der Hauptunterschied liegt darin, daß *Write* die erzeugten Zeichen danach ausgibt, während *Str* sie in einem String als Funktionsergebnis zurückliefert.

Querverweis *Val, Write*

Beispiel

```
function itoa(i : LongInt): string;
  { verwandelt von »Integer« nach »ASCII« }
var
  s : string[11];
begin
  Str(i,s);
  itoa := s;
end;

begin
  Writeln('Die folgenden beiden Ausgaben sollten',
          'wirklich gleich sein:');
  Writeln(-5322, ' <---> ', itoa(-5233));
end.
```

Succ-Funktion

Funktion liefert den Nachfolger des Arguments zurück.

Deklaration `Succ(x)`

Ergebnistyp derselbe wie der des Arguments.

Erläuterung *x* ist ein ordinaler Ausdruck beliebigen Typs. Das Ergebnis der Funktion hat denselben Typ und stellt den Nachfolger von *x* dar.

Grenzen — wenn *x* das Element mit dem höchsten ordinalen Wert innerhalb seines Typs darstellt, dann ist der Nachfolger von *x* nicht definiert und *Succ* erzeugt einen Laufzeitfehler.

Querverweis — *Dec*, *Inc*, *Pred*

Swap-Funktion

Funktion — vertauscht das niederwertige und das höherwertige Byte des Arguments miteinander.

Deklaration — `Swap(x)`

Ergebnistyp — derselbe Typ wie der des Arguments.

Erläuterung — *x* ist ein Ausdruck des Typs *Integer* oder *Word*.

Querverweis — *Hi*, *Lo*

Beispiel

```
var
  x : Word;
begin
  x := Swap($1234);   { x := $3412 }
end.
```

SwapVectors-Prozedur Dos

Funktion — vertauscht die von *System* belegten Interrupt-Vektoren mit den entsprechenden Variablen der Unit *System*.

Deklaration — `SwapVectors`

Erläuterung — *SwapVectors* tauscht die Interrupt-Vektoren $00, $02, .. $75 und $34..$3E mit den Inhalten der globalen Variablen *SaveInt00*..*SaveInt75* bzw. *SaveInt34*..*SaveInt3E*: Der erste Aufruf setzt diese Vektoren auf die Werte zurück, die sie vor dem Start des Programms hatten, der zweite setzt wieder die Routinen der Laufzeitbibliothek ein usw.

SwapVectors sollte vor und nach der Ausführung eines anderen Programms via *Exec* eingesetzt werden. Damit lassen sich sozusagen zwei Fliegen mit einer Klappe schlagen:

- das aufgerufene Programm kann mit den DOS-Originalvektoren arbeiten (und endet beispielsweise bei Fehlern nicht in der Turbo-Laufzeitbibliothek);
- eventuelle Veränderungen von Vektoren, die für das Turbo Pascal-Programm lebenswichtig sind (hauptsächlich: $34..$3E bei Einsatz des Emulators), werden durch den zweiten Aufruf von *SwapVectors* (nach *Exec*) rückgängig gemacht.

Querverweis — *Exec*

Beispiel

```
program SwapVectorsDemo;
{$M 8192,0,0}   { kleiner Stack, kein Heap }
uses Dos;
var  CmdLine: string[79];

begin
  Write('Geben Sie den DOS-Befehl ein: ');
  Readln(CmdLine);

  if CmdLine <> '' then CmdLine := '/C' + CmdLine;
  SwapVectors;        { Originalvektoren setzen }
  Exec(GetEnv('COMSPEC'), CmdLine); { Ausführung }
  SwapVectors;       { Turbo-Vektoren wiederherstellen }
  if DosError <> 0 then
    Writeln('Befehl wurde nicht korrekt ausgeführt!');
end.
```

TextBackground-Prozedur Crt

Funktion legt die Hintergrundfarbe für folgende Textausgaben fest.

Deklaration `TextBackground(Color: Byte)`

Erläuterung *Color* ist ein Ausdruck des Typs *Byte*. Mögliche Werte beschränken sich auf den Bereich 0..7 und entsprechen den folgenden (in *Crt* definierten) Farb-Konstanten:

```
const
{ Für Vorder- und Hintergrund }
  Black        = 0;   { Schwarz }
  Blue         = 1;   { Blau }
  Green        = 2;   { Grün }
  Cyan         = 3;   { Türkis }
  Red          = 4;   { Rot }
  Magenta      = 5;   { Fuchsinrot }
  Brown        = 6;   { Braun }
  LightGray    = 7;   { Hellgrau }
```

Die Unit *Crt* deklariert eine globale Variable namens *TextAttr*, die das jeweils verwendete Zeichenattribut enthält. Ein Aufruf von *TextBackground* setzt die Bits 4..6 dieser Variablen mit der als *Color* angegebenen Farbe.

Querverweis *TextColor*, *HighVideo*, *NormVideo*, *LowVideo*.

TextColor-Prozedur Crt

Funktion legt die Zeichenfarbe für folgende Textausgaben fest.

Deklaration `TextColor(Color: Byte)`

Erläuterung *Color* ist ein Ausdruck des Typs *Byte*. Mögliche Werte dieses Parameters beschränken sich auf den Bereich von 0..15 und entsprechen den folgenden (in der Unit *Crt* definierten) Farb-Konstanten:

```
const
{ Für Vorder- und Hintergrund }
  Black        = 0;   { Schwarz }
  Blue         = 1;   { Blau }
  Green        = 2;   { Grün }
  Cyan         = 3;   { Türkis }
  Red          = 4;   { Rot }
  Magenta      = 5;   { Fuchsinrot }
  Brown        = 6;   { Braun }
  LightGray    = 7;   { Hellgrau }

{ Nur für Vordergrundfarben }
  DarkGray     = 8;   { Dunkelgrau }
  LightBlue    = 9;   { Hellblau }
  LightGreen   = 10;  { Hellgrün }
  LightCyan    = 11;  { helles Türkis }
  LightRed     = 12;  { Hellrot }
  LightMagenta = 13;  { helles fuchsinrot }
  Yellow       = 14;  { Gelb }
  White        = 15;  { Weiß }

  Blink    = 128;  { blinkend -- siehe unten }
```

Die Unit *Crt* deklariert eine globale Variable namens *TextAttr*, die das jeweils verwendete Zeichenattribut enthält. Ein Aufruf von *TextColor* setzt die Bits 0-3 dieser Variablen mit der als *Color* angegebenen Farbe.

Durch die Addition der Konstanten *Blink* zum Wert von *Color* wird das Bit 7 von *TextAttr* zusätzlich gesetzt: folgende Textausgaben erscheinen blinkend.

Querverweis *HighVideo*, *LowVideo*, *NormVideo*, *TextBackground*.

Beispiel

```
  ...
TextColor(Green);      { grüne Zeichen }
TextColor(Red+Blink);  { rote blinkende Zeichen }
TextColor(31);         { weiße blinkende Zeichen }
TextColor(47);         { dito }
  ...
```

TextHeight-Funktion Graph

Funktion liefert die Höhe eines Textstrings (in Pixel) zurück.

Deklaration `TextHeight(TextString : string)`

Ergebnistyp *Word*

Erläuterung *TextHeight* berechnet aus dem übergebenen String sowie dem momentan gesetzten Zeichensatz und dem Vergrößerungsfaktor den vertikalen Platzbedarf einer Textzeile. Wenn beispielsweise der Standard-Zeichensatz (8 * 8 Pixel) zusammen mit dem Vergrößerungsfaktor 1 gesetzt ist, dann liefert *TextHeight* für den String *Turbo* das Ergebnis 8 zurück; ist der Vergrößerungsfaktor 2 gesetzt, dann hat das Ergebnis den Wert 16 usw.

Hinweis: Im Gegensatz zur Funktion *TextWidth* spielt es bei *TextHeight* keine Rolle, welcher Text zur Berechnung übergeben wird.

Der Platzbedarf von Strings sollte *grundsätzlich* über die Funktionen *TextHeight* und *TextWidth* ermittelt werden. Nicht nur, daß das Zentrieren von Titeln, Einpassen von Beschriftungen usw. auf diese Weise wesentlich einfacher geht – Programme, die diese Funktionen anstelle von Konstanten verwenden, erfordern bei einem Wechsel der Schriftart oder des Grafik-Treibers erheblich weniger Modifikationen (wenn überhaupt).

Grenzen vor dem Aufruf von *TextHeight* muß ein Grafikmodus gesetzt worden sein.

Querverweis *OutText, OutTextXY, SetTextStyle, SetUserCharSize, TextWidth.*

Beispiel

```
program TextHeightDemo;
uses Graph;
var
  GraphMode, GraphDriver: Integer;
  x,y : Integer;

begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');
  y := 0; SetTextJustify(LeftText, BottomText);
  for x:= DefaultFont to GothicFont do
  begin
    SetTextStyle(x,0,7);
    if y = 0 then y := TextHeight('T') + 5
             else y:= y + TextHeight('T') + 5;
    OutTextXY(0,y,'Turbo Pascal!!');
  end;
  Readln;
  CloseGraph;
end.
```

TextMode-Prozedur — Crt

Funktion — setzt einen Textmodus.

Deklaration

```
TextMode(Mode : Word)
```

Erläuterung — *Mode* ist ein Word-Ausdruck und legt den zu setzenden Textmodus fest. Die Unit *Crt* definiert die folgenden Konstanten für Textmodi:

```
const   { Textmodi }
  BW40  = 0;    { 40 * 25 schwarzweiß, CGA }
  CO40  = 1;    { 40 * 25 farbig, CGA }
  BW80  = 2;    { 80 * 25 schwarzweiß, CGA }
  CO80  = 3;    { 80 * 25 farbig, CGA }
  Mono  = 7;    { 80 * 25 s/w , Monochrom-Adapter }

  Font8x8 = 256; { für 43/50 Zeilen auf EGA/VGA }

  C40 = CO40;    { wg. Kompatibilität zur Version 3.0 }
  C80 = CO80;    { dito }
```

Für alle anderen Werte setzt *TextMode* den Wert *CO80* als Standardvorgabe. Der Aufruf von *TextMode* mit der Variablen *LastMode* als Parameter setzt den Textmodus wieder, der vor der letzten Umschaltung zur Grafik bzw. dem letzten Aufruf von *TextMode* aktiv war (und macht eigentlich nur Sinn, wenn nicht nur mit *Crt*, sondern auch zusätzlich mit *Graph* oder *Graph3* gearbeitet wird).

Mit der Addition von *Font8x8* zu einer der Konstanten beim Aufruf von *TextMode* lassen sich 43 Textzeilen (EGA) bzw. 50 Textzeilen (VGA) setzen:

```
TextMode(C80 + Font8x8);
```

Dieser Aufruf setzt auf EGA-Karten »80*43 Zeichen farbig«, auf VGA-Karten »80*50 Zeilen farbig«. Da auch dieser Modus in *LastMode* gespeichert wird, muß für eine *Umschaltung* der Zeilenzahl (unter Beibehaltung des momentan gesetzten Videomodus) das höherwertige Byte dieser Variablen ausgeblendet werden:

```
TextMode(Lo(LastMode) + Font8x8); { schaltet auf 43
                                    bzw. 50 Zeilen um }
TextMode(Lo(LastMode));  { zurück auf 25 Zeilen Text }
```

Hinweis: Wenn bereits vor dem Start eines Turbo Pascal-Programms ein Textmodus mit 43 bzw. 50 Zeilen gesetzt war, wird dieser Modus in *LastMode* gespeichert und die untere rechte Fenstergrenze (*WindMax*) entsprechend gesetzt.

Jeder Aufruf von *TextMode* setzt das Textfenster auf die volle Bildschirmgröße zurück, die Variable *DirectVideo* auf *True* und

die Variable *CheckSnow* ebenfalls auf *True*, wenn ein farbiger Modus gewählt wurde. Das Textattribut für folgende Ausgaben (*TextAttr*) wird auf »normal« gesetzt, was einem Aufruf der Prozedur *NormVideo* entspricht.

Grenzen — Grafikmodi werden von *TextMode* nicht unterstützt (und nicht in *LastMode* gespeichert) – mit dieser Routine lassen sich nur zuvor gesetzte Textmodi wiederherstellen.

Querverweis — *RestoreCrtMode*, *SetGraphMode*.

TextWidth-Funktion — Graph

Funktion — liefert die Breite eines Textstrings zurück.

Deklaration — `TextWidth(TextString : string)`

Ergebnistyp — *Word*

Erläuterung — *TextWidth* berechnet aus dem übergebenen String sowie dem momentan gesetzten Zeichensatz und dem Vergrößerungsfaktor den horizontalen Platzbedarf einer Textzeile. Wenn beispielsweise der Standard-Zeichensatz (8 * 8 Pixel) zusammen mit dem Vergrößerungsfaktor 1 gesetzt ist, dann liefert *TextWidth* für den String *Turbo* das Ergebnis 40 (=5 * 8) zurück; ist der Vergrößerungsfaktor 2 gesetzt, dann hat das Ergebnis den Wert 80 usw.

Hinweis: Im Gegensatz zur Funktion *TextHeight* spielt es hier sehr wohl eine Rolle, aus wievielen (und bei proportionalen Zeichensätzen: welchen) Zeichen sich der Text zusammensetzt, der der Funktion zur Berechnung übergeben wird.

Der Platzbedarf von Strings sollte *grundsätzlich* über die Funktionen *TextHeight* und *TextWidth* ermittelt werden. Nicht nur, daß das Zentrieren von Titeln, Einpassen von Beschriftungen usw. auf diese Weise wesentlich einfacher geht – Programme, die diese Funktionen anstelle von Konstanten verwenden, erfordern bei einem Wechsel der Schriftart oder des Grafik-Treibers erheblich weniger Modifikationen (wenn überhaupt).

Grenzen — vor dem Aufruf von *TextWidth* muß ein Grafikmodus gesetzt worden sein.

Querverweis — *OutText*, *OutTextXY*, *SetTextStyle*, *TextHeight*, *SetUserCharSize*.

Beispiel

```
program TextWidthDemo;
uses Graph;
const
  MsgStr : string = 'Turbo Nasskalt!';
var
  GraphMode, GraphDriver: Integer;
```

```
  x : Integer;
  OurFont: Integer;

begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '\TP');

  OurFont := GothicFont;
  SetTextJustify(CenterText, CenterText);
  x := 1;
  repeat
    SetTextStyle(OurFont, HorizDir, x);
    SetColor(x);
    OutTextXY(GetMaxX div 2, (GetMaxY div 2) + 8 * x,
              MsgStr);
    Inc(x);
  until (x = 11) or (TextWidth(MsgStr) > GetMaxX);
  Readln;
  CloseGraph;
end.
```

Trunc-Funktion

Funktion	wandelt einen Realwert durch Abschneiden der Nachkommastellen in einen Integerwert um.
Deklaration	`Trunc(x: Real)`
Ergebnistyp	*LongInt*
Erläuterung	*x* ist ein Real-Ausdruck. Das Ergebnis der Funktion ist der ganzzahlige Anteil von *x*.
Grenzen	wenn der Wert von *x* außerhalb des Rechenbereichs von *LongInt* liegt, dann erzeugt *Trunc* einen Laufzeitfehler.
Querverweis	*Int, Round*

Truncate-Prozedur

Funktion	schneidet eine Datei an der momentanen Position ab.
Deklaration	`Truncate(var f)`
Erläuterung	*f* steht für eine Datei-Variable beliebigen Typs (außer *Text*). Die mit *f* verbundene Datei wird an der Stelle abgeschnitten, auf die der Positionszeiger zum Zeitpunkt des Aufrufs von *Truncate* zeigt. Zwischen dieser Position und dem physikalischen Dateiende befindliche Daten werden gelöscht, die Funktion *Eof(f)* liefert nach der Ausführung von *Truncate* das Ergebnis *True*.

Wenn *Truncate* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus `{$I-}` den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Grenzen die durch *f* bezeichnete Datei muß offen sein. *Truncate* läßt sich nicht auf Text-Dateien anwenden.

Querverweis *Seek*, *Reset*, *Rewrite*

TypeOf-Funktion

Funktion liefert einen Zeiger auf die virtuelle Methodentabelle (VMT) eines Objekttyps.

Deklaration `TypeOf( x)`

Erläuterung Der Parameter *x* kann entweder die Bezeichnung eines Objekttyps oder eine Instanz eines Objekttyps. In beiden Fällen ist das Ergebnis vom Typ *Pointer* ein Zeiger auf die virtuelle Methodentabelle (VMT) des Objekttyps. *TypeOf* darf nur auf Objekttypen mit einer VMT angewendet werden. Bei allen anderen Datentypen liefert die Funktion einen Fehler. Über *TypeOf* kann zum Beispiel der tatsächliche Datentyp einer Instanz überprüft werden:

```
if TypeOf(Self) = TypeOf(Point) then ...
```

UnpackTime-Prozedur Dos

Funktion konvertiert Datum und Uhrzeit aus einem gepackten Format in einen Record des Typs *DateTime*.

Deklaration `UnpackTime(Time: LongInt; var DT: DateTime)`

Erläuterung »gepackte« *LongInt*-Variablen, die ein Kalenderdatum und eine Uhrzeit zusammen in 4 Bytes speichern, werden von den Routinen *GetFTime* und *PackTime* sowie von *FindFirst* und *FindNext* erzeugt. *UnpackTime* »entpackt« einen solchen *LongInt* und setzt die Felder des als *DT* übergebenen Records mit den einzelnen Werten. Der Datentyp *DateTime* ist in der Unit *Dos* folgendermaßen definiert:

```
DateTime = record
    Year, Month,    { Jahr: 1980..2099, Monat 1..12 }
    Day, Hour,      { Tag: 1..31, Stunde 0..23 }
    Min, Sec : Word;  { jeweils 0..59 }
  end;
```

Grenzen die einzelnen Bit-Felder des als *Time* übergebenen Wertes werden nicht geprüft – nötigenfalls muß das Programm direkt

verifizieren, daß sich ihre Werte in den zugelassenen Grenzen bewegen.

Querverweis *PackTime*, *GetFTime*, *SetFTime*, *GetTime*, *SetTime*

UpCase-Funktion

Funktion verwandelt Klein- in Großbuchstaben.

Deklaration `UpCase(ch: Char)`

Ergebnistyp *Char*

Erläuterung *ch* ist ein Ausdruck des Typs *Char*. Wenn *ch* einen Kleinbuchstaben im Bereich von 'a'..'z' darstellt, liefert *UpCase* den entsprechenden Großbuchstaben zurück.

Val-Prozedur

Funktion interpretiert einen String als numerischen Wert.

Deklaration `Val( s: string ; v; var code: Integer)`

Erläuterung *s* ist der zu interpretierende String-Ausdruck, *v* ist eine Variable mit einem Integer- oder Real-Typ und bekommt das Ergebnis der Interpretation zugewiesen. In der als *code* übergebenen Integer-Variablen speichert *Val* bei einer erfolgreichen Konvertierung den Wert 0 – ansonsten steht hier die Position des Zeichens in *s*, durch das die Interpretation als fehlerhaft abgebrochen wurde.

s muß eine Zeichenfolge enthalten, die den in Kapitel 1 »Sprachelemente und Konstanten« des *Programmierhandbuches* aufgeführten Regeln genügt.

Führende Leerzeichen und Tabs in *s* werden von *Val* automatisch übersprungen und müssen deshalb nicht eigens entfernt werden:

```
Val('   123', Int_Var, code);   { ergibt 123 }
Val('123', Int_Var, code); { dito }
```

Grenzen das Verhalten von *Val* bei Überläufen hängt vom Stand des Compiler-Schalters `{$v}` (Bereichs-Prüfung) und dem Typ des übergebenen Parameters *v* ab. Im Modus `{$v+}` erzeugt *Val* einen Laufzeitfehler, wenn das Ergebnis der Interpretation nicht im zulässigen Wertebereich von *v* liegt, also zum Beispiel:

```
Val('999999', Int_Var, code);  { würde Int_Var den Wert
                                 999999 zuordnen }
```

Im Modus `{$v-}` erzeugt *Val* bei Überläufen für Real-Typen und *LongInt* ein undefiniertes Ergebnis, wobei *code* in jedem Fall auf einen Wert <>0 gesetzt wird. Für alle anderen Varia-

blentypen wird in *v* der »Überlauf-Rest« gespeichert, *code* bekommt den Wert 0 (!). Aus diesem Grund sollte für die Konvertierung von Integerwerten immer ein *LongInt* benutzt und die Bereichsprüfung vorher abgeschaltet werden:

```
{$R-}
Val('999999', LongInt_Var, code);
if (code <> 0> or (LongIntVar < -32767) or
    (LongIntVar > 32768) then ....          { Fehler! }
  else
    Int_Var := LongIntVar;
```

In diesem Beispiel wird nach der Konvertierung explizit geprüft, ob das Ergebnis im zulässigen Wertebereich (hier: für *Integer*) liegt.

Querverweis *Str*

Beispiel

```
var
  i, code: Integer;
begin
  { Interpretiert den 1. Kommandozeilen-
    Parameter als numerischen Wert }
  Val(ParamStr(1), i, code);
  if code <> 0 then
   Writeln('Kann nicht numerisch interpretiert werden');
  else
   Writeln('numerischer Wert: ', i);
end.
```

WhereX-Funktion Crt

Funktion liefert die Spaltenposition des Cursors (relativ zum momentanen Fenster) zurück.

Deklaration `WhereX`

Ergebnistyp *Byte*

Erläuterung die Zählung beginnt mit 1: wenn der Cursor beispielsweise mit *ClrScr* in die obere linke Fensterecke gesetzt wurde, dann liefert ein folgender Aufruf von *WhereX* den Wert 1.

Querverweis *GotoXY, WhereY, Window.*

WhereY-Funktion Crt

Funktion liefert die Zeilenposition des Cursors (relativ zum momentanen Fenster) zurück.

Deklaration `WhereY`

Ergebnistyp *Byte*

Erläuterung die Zählung beginnt mit 1: wenn der Cursor beispielsweise mit *ClrScr* in die obere linke Fensterecke gesetzt wurde, dann liefert ein folgender Aufruf von *WhereY* den Wert 1.

Querverweis *GotoXY*, *WhereX*, *Window*.

Window-Prozedur Crt

Funktion definiert einen Bereich des Bildschirms als Textfenster.

Deklaration
```
Window(x1,y1,x2,y2: Byte)
```

Erläuterung alle vier Parameter dieser Prozedur sind Ausdrücke des Typs *Byte*; (*x1*,*y1*) legen die obere linke und (*x2*,*y2*) die untere rechte Ecke des Fensters fest. Die Grenzen des momentan gesetzten Fensters werden von *Window* in den globalen *Word*-Variablen *WindMin* und *WindMax* gespeichert.

Die obere linke Ecke des Bildschirms hat die Koordinaten (1,1), die Minimalgröße eines Fensters ist ein einzelner Buchstabe (also eine Spalte und eine Zeile). Wenn auch nur einer der vier Parameter von *Window* nicht innerhalb der erlaubten Grenzen liegt, wird der Prozeduraufruf ignoriert, das momentan gesetzte Fenster bleibt unverändert.

In den Videomodi mit 80 Spalten verwendet Turbo Pascal standardmäßig ein Fenster, das einem Aufruf von *Window* mit den Parametern (1,1,80,25) (bzw. (1,1,80,43) für EGA-Karten im Modus mit 43 und (1,1,80,50) für VGA-Karten im Modus mit 50 Textzeilen) entspricht; in den Videomodi mit 40 Spalten haben diese Parameter die Werte (1,1,40,25).

Sämtliche Bildschirm-Koordinaten außer den Parametern von Window sind relativ zum momentan gesetzten Textfenster – wohin der Cursor beispielsweise durch einen Befehl wie *GotoXY*(*1*,*1*) innerhalb des Bildschirms gesetzt wird, ist ausschließlich von der Position dieses Fensters abhängig.

Hinweis: Bei einer fehlerfreien Ausführung setzt *Window* den Cursor auf die relativen Koordinaten (1,1) innerhalb des neuen Fensters, d.h. in seine obere linke Ecke.

Weitere Details finden Sie in Kapitel 12 »Standard-Units« des *Programmierhandbuchs*.

Querverweis *SetViewPort*

Beispiel
```
program WindowDemo;
uses Crt;
var  x, y: Byte;
begin
  TextBackground(Black); ClrScr;
  repeat
```

```
    x:= Succ(Random(70)); y := Succ(Random(17));
    Window(x,y,x+Random(10),y+Random(8));
    TextBackground(Random(16));
    ClrScr; Delay(50);
  until KeyPressed;
end.
```

Write-Prozedur

Funktion schreibt Daten in eine Datei.

Deklaration
```
Write(var f, v1 [, v2,...,vn])        für typisierte Dateien
Write([var f: Text;] v1 [,v2,...,vn])  für Textdateien
```

Erläuterung *Write für typisierte Dateien: f* steht für eine Datei-Variable beliebigen Typs (außer *Text*), jede der angegebenen Variablen *v* hat denselben Typ wie die Komponenten dieser Datei. Der Positionszeiger innerhalb der Datei wird durch jede Schreibaktion um eine Komponente weiterbewegt. Wenn der Positionszeiger vor einer Schreibaktion auf dem Dateiende steht (d.h. *Eof(f)* das Ergebnis *True* hat), wird die Datei durch die Schreibaktion vergrößert. Da die momentane Position immer auf die jeweils nächste Komponente gesetzt wird, liefert *Eof(f)* in diesem Fall auch nach der Schreibaktion weiterhin das Ergebnis *True*.

Wenn *Write* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus `{$I-}` den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Write für Textdateien: wenn *f* angegeben ist, muß dieser Parameter für eine Textdatei-Variable stehen – ist *f* nicht angegeben, dann schreibt *Write* zur Standard-Ausgabe (*Output*).

Jede Angabe *v* in der Syntaxdefinition von *Write* steht für einen Ausdruck (OutExpr), dessen Wert in die Datei geschrieben wird. Jedem Ausdruck kann die Angabe einer Feldbreite folgen, für Ausdrücke mit Real-Typen ist zusätzlich die Angabe der gewünschten Anzahl von Nachkommastellen möglich. Jeder Ausgabe-Ausdruck muß entweder ein Char-Typ, ein Integer-Typ, ein Real-Typ, ein *String* oder *Boolean sein.*

Die Parameter für *Write* werden folgendermaßen notiert:

```
OutExpr [ : MinBreite [ : DezStellen]]
```

MinBreite und *DezStellen* sind Integer-Ausdrücke. *MinBreite* definiert die minimale Breite des Feldes und muß größer als 0 sein. Es werden mindestens soviele Zeichen ausgeschrieben, wie durch *MinBreite* vorgegeben sind; enthält die Zeichenkette weniger Zeichen, werden entsprechend viele Leerzeichen vorangestellt. *OutExpr* wird durch eine zu niedrige Angabe von

MinBreite nicht abgeschnitten: es werden immer mindestens so viele Zeichen ausgegeben, wie zur Darstellung von *OutExpr* nötig sind. Dasselbe gilt, wenn *MinBreite* nicht angegeben ist.

DezStellen definiert die Anzahl der Dezimalstellen in einer Fixkomma-Darstellung eines Real-Typs und kann nur angegeben werden, wenn *OutExpr* einen Real-Typ hat und *MinBreite* ebenfalls definiert ist. *DezStellen* ist ein Integer-Ausdruck und muß einen Wert größer Null haben.

Write mit einem Char-Typen: ist *MinBreite* nicht angegeben, wird das durch *OutExpr* dargestellte Zeichen in die Datei geschrieben, ansonsten werden *MinBreite-1* Leerzeichen vorangestellt.

Write mit einem Integer-Typen: wenn *MinBreite* nicht angegeben ist, wird die dezimale Darstellung von *OutExpr* ohne führende Leerzeichen in die Datei geschrieben. Ist *MinBreite* definiert und dieser Wert größer als die Länge der Ziffernfolge, werden entsprechend viele Leerzeichen vorangestellt.

Write mit einem Real-Typen: wenn *OutExpr* einen Real-Typ hat, wird der entsprechende Wert in dezimaler Darstellung in die Datei geschrieben. Das Format dieser Darstellung wird über den Wert von *DezStellen* bzw. durch die Abwesenheit dieses Parameters bestimmt.

Wenn *DezStellen* nicht angegeben ist, wird die Gleitkomma-Notation benutzt. Ist auch *MinBreite* nicht angegeben, dann benutzt Turbo Pascal den Wert 10 als Standardvorgabe. Angaben für *MinBreite* außerhalb des Bereichs von 10..80 haben dieselbe Wirkung wie eine Angabe des Wertes 10. Das Format eines dezimalen Strings ist:

```
[ - ] <Ziffer> . <Nachkst.> e [ + | - ] <Exponent>
```

Die einzelnen Komponenten dieses Diagramms haben folgende Bedeutungen:

Komponente	Bedeutung
`[ - ]`	Nichts oder »-«, entsprechend dem Vorzeichen von *OutExpr*
`<Ziffer>`	Einzelne Ziffer, nur »0«, wenn auch *OutExpr* 0 ist
.	Dezimalpunkt
`<Nachkst.>`	Ziffernfolge mit *MinBreite* -9 Ziffern
`e`	Kleingeschriebenes »e«
`[ + | - ]`	»+« oder »-«, entsprechend dem Vorzeichen des Exponenten
`<Exponent>`	Exponent mit nachfolgenden Leerzeichen, wenn weniger als 4 Stellen

Wenn *DezStellen* angegeben ist, wird ein String in Fixkomma-Notation geschrieben, der folgendes Format hat:

```
[ <Leerz.> ] [ - ] <Ziffern> [ . <Nachkst.> ]
```

Die einzelnen Komponenten dieses Diagramms haben folgende Bedeutung:

Komponente	Bedeutung
`[ <Leerz.> ]`	Leerzeichen zur Angleichung an die Vorgabe von *MinBreite*
`[-]`	»-«, wenn *OutExpr* negativ ist
`<Ziffern>`	mindestens eine Ziffer, keine führenden Nullen
.	Dezimalpunkt (wenn *DezStellen* >0)
`<Nachkst.>`	Nachkommastellen (wenn *DezStellen* >0)

Wenn *DezStellen* angegeben und *OutExpr* größer oder gleich 10^ (19 - *DezStellen*) ist, wird die Zahl in Gleitkomma-Notation mit 19 signifikanten Ziffern formatiert, wobei entsprechend dem Wert von *MinBreite* Leerzeichen vorangestellt werden.

Write mit einem String-Typen: ist *MinBreite* nicht angegeben, wird der Stringwert von *OutExpr* ohne führende Leerzeichen in die Datei geschrieben. Wenn *MinBreite* angegeben und dieser Wert größer als die Länge von OutExpr ist, wird eine entsprechende Anzahl von Leerzeichen vorangestellt.

Write mit einem Boolean: Wenn *OutExpr* den Typ *Boolean* hat, repräsentiert *Write* den Wert durch die Zeichenfolgen *True* oder *False*, die wie Strings entsprechender Länge behandelt werden.

Wenn *Write* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus `{$I-}` den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<>0) als Ergebnis.

Grenzen die durch *f* bezeichnete Datei muß offen sein.

Querverweis *Str, Writeln*

Beispiel

```
program WriteDemo;
type
  Datensatz = record
      Name : string[20];
      Alter: Byte;
    end;
var
  FTyp : file of Datensatz;  { typisierte Datei }
  x,y,z : Integer;
```

```
  r : Real;
  EinSatz: Datensatz;
begin
{  .... }
  for x := 1 to 20 do
    Write(FTyp, EinSatz);  { schreibt jeweils
                             einen ganzen Record }

  x:= 10; y := -24857; z:= 0;
  Writeln('!', x:4, y, z:8, '!'); { !  10-24857     0! }
  r := 49349857.0;
  Writeln(r);                 {  4.9349857000E+07     }
  Writeln('!',r:4,'!');       { ! 4.9E+07!            }
  Writeln('!',r:10,'!');      { ! 4.935E+08!          }
  Writeln('!',r:10:0,'!');    { !  49349857!          }
  Writeln('!',r:12:6,'!');    { !49349856.999900!     }
end.
```

Writeln-Prozedur

Funktion	schreibt Daten in eine Textdatei und schließt mit einem Zeilenvorschub ab.
Deklaration	`Writeln([var f: Text;] v1 [,v2,...,vn])`
Erläuterung	*Writeln* ist nur auf Textdateien anwendbar und behandelt angegebene Parameter auf dieselbe Art wie *Write*. Der einzige Unterschied besteht darin, daß *Writeln* den Schreibvorgang mit der Ausgabe eines Zeilenvorschubs beendet. *Writeln*(*f*) ohne weitere Parameter schreibt ein Zeilenende-Zeichen in die durch *f* angegebene Datei; *Writeln* ohne Angabe einer Datei-Variablen entspricht *Writeln*(*Output*)und erzeugt einen Zeilenvorschub auf dem Bildschirm.
Grenzen	die durch *f* bezeichnete Datei muß offen sein und den Typ *Text* haben.
Querverweis	*Str, Write*

2. Compiler-Befehle

Wie der Name vermuten läßt, weist ein Compiler-Befehl den Compiler während der Übersetzung eines Quelltextes an, bestimmte Operationen auszuführen (oder zu unterlassen). In Turbo Pascal gibt es mehrere Möglichkeiten, Compiler-Befehle zu formulieren:

- als Kommentar mit einer speziellen Syntax innerhalb des Quelltextes: direkt auf die öffnende Klammer (`(*` oder `{`) folgen ein Dollarzeichen (`$`) und der Name des Befehls, der aus einem oder mehreren Buchstaben besteht. Diese »Kommentare« können – von einigen Ausnahmen abgesehen – an jeder Stelle des Quelltextes erscheinen, an der auch ein normaler Kommentar zulässig ist;
- über einen Wahlpunkt des Menüs *Options/Compiler*;
- als Kommandozeilen-Parameter im Format `/$<Befehl>` bzw. `/<Parameter>`.

Compiler-Befehle lassen sich in drei Klassen einteilen: Schalter, Parameter und Bedingungen.

- Mit einem *Schalter* werden bestimmte Funktionen des Compilers ein- oder ausgeschaltet. Dementsprechend ist die Syntax: auf den Namen des Befehls folgt entweder ein Minuszeichen (`-`) oder ein Pluszeichen (`+`). Für jeden Schalter existiert eine Standardvorgabe; ein einmal gesetzter Schalter bleibt solange aktiv, bis er durch einen erneuten Compiler-Befehl wieder zurückgesetzt wird (oder bis zum Ende der Compilierung).

- Ein *Parameter* legt Werte (wie die Größe des Stacks) und Dateinamen fest. Auch hier wird in den meisten Fällen mit Standardvorgaben gearbeitet, wenn Sie keine eigenen Angaben machen. Im Gegensatz zu Schaltern muß immer ein Leerzeichen zwischen dem Befehlsnamen und dem folgenden Parameter stehen.

- Eine *Bedingung* kontrolliert, welche Teile eines Quelltextes compiliert werden (bedingte Compilierung). Für Compiler-Befehle dieser Art gibt es keine Standardvorgaben: normalerweise bearbeitet der Compiler den gesamten Inhalt einer Quelltext-Datei. Auch hier muß zwischen dem Befehl und der Bedingung mindestens ein Leerzeichen stehen.

Bevor wir auf die Einzelheiten eingehen, erst einmal einige Beispiele für gültige Compiler-Befehle:

```
{$B+}                              { vollständige Auswertung
                                     boolscher Ausdrücke }
{$R- - Bereichsüberprüfung aus }   { Kommentar im Befehl! }
{$I D_TYPEN.INC}                   { Include-Datei D_TYPEN.INC }
{$O EdFormat}                      { Aufnahme einer Unit, Deklara-}
                                   { tion dieser Unit als Overlay }
{$M 65520,8192,655360}             { 64 KByte Stack, mindestens
                                     8 KByte Heap, max. 640 KB Heap }
{$DEFINE Debug}                    { Definition des Symbols Debug }
{$IFDEF Debug}                     { Wenn Debug definiert ist, dann }
  ...                              { wird dieser Teil compiliert }
  ...                              { - ansonsten nicht }
{$ENDIF}                           { Ende der bedingten Compilierung }
```

Schalter

Compiler-Schalter arbeiten entweder *global* (für die gesamte Compilierung) oder *lokal* (bis zu einer erneuten Umschaltung). Globale Schalter müssen grundsätzlich nach dem einleitenden **program** oder der **Unit** und vor jedem anderen Teil des Programms stehen (d.h. vor dem ersten **uses**, **label**, **const**, **type**, **procedure**, **function** oder **begin**). Lokale Schalter können dagegen an beliebiger Stelle des Programms erscheinen.

Turbo Pascal erlaubt es, mehrere Schalter in einer Kommentarklammer zusammenzufassen, solange sie durch Kommas voneinander getrennt sind. Das Dollarzeichen erscheint in diesem Fall nur einmal, Leerzeichen oder »echte« Kommentare *zwischen* den Befehlen sind nicht erlaubt. Einige Beispiele:

```
{$B+,R-,S+}
{$B+,R-,S+    <-- dieser Kommentar ist legal }
{$B+,R-, S+   <-- ergibt eine Fehlermeldung }
{$B+,R- S+    <-- "S+" wird als "echter" Kommentar ignoriert }
```

Ausrichtung von Variablen im Speicher

Syntax:	{$A+} oder {$A-}bzw. /$A+ oder /$A+
Standard:	{$A+}
Typ:	global
Menü-Äquivalent:	*Options/Compiler/Word align data*

Dieser Schalter legt fest, wie Compiler und Linker Variablen mit einem Platzbedarf von mehr als einem Byte im Speicher anordnen.

Im Modus {$A+} (Word) werden alle Variablen außer *Char* und *Byte* so im Speicher angeordnet, daß der durch sie belegte Bereich mit einer geradzahligen Adresse beginnt. Zugriffe des 8086 und seiner Nachfolger werden dadurch beschleunigt; bei einem 8088, der nur jeweils 8 Bit auf einmal lesen kann, hat die Anordnung der Variablen keine weiteren Wirkungen.

Im Modus {$A-} (Byte) findet keine spezielle Ausrichtung statt – Variablen werden hier so dicht wie möglich hintereinander gepackt.

Hinweis: **var**- und **const**-Deklarationsabschnitte werden – unabhängig vom Stand des Schalters {$A} – vom Linker grundsätzlich mit einer geraden Speicheradresse begonnen. Analoges gilt für den Stackzeiger (Prozessor-Register SP), der immer eine gerade Adresse enthält: beim Belegen von lokalem Speicherplatz fügt der Compiler nötigenfalls ein »Leerbyte« hinzu.

Auswertung boolscher Ausdrücke

Syntax:	{$B+} oder {$B-} bzw. /$B+ oder /$B-
Standard:	{$B-}
Typ:	lokal
Menü-Äquivalent:	*Options/Compiler/Complete boolean evaluation*

Dieser Schalter legt fest, wie mit *and* und *or* verbundene boolsche Ausdrücke ausgewertet werden.

Im Modus {$B+} (Complete) erzeugt der Compiler Code für eine vollständige Auswertung – sämtliche Teile eines zusammengesetzten Ausdrucks werden auch dann berechnet, wenn das Ergebnis des Gesamtausdrucks bereits feststeht.

Im Modus {$B-}(Short Circuit) wird ein *Kurzschlußverfahren* benutzt: die Teile eines Ausdrucks werden grundsätzlich in der Reihenfolge ihrer Angabe ausgewertet; die Auswertung bricht ab, sobald sich am Gesamtergebnis nichts mehr ändern kann (für **or**: ein Teilausdruck ergibt *True*, für **and**: ein Teilausdruck ergibt *False*). Beispiele für die Unterschiede zwischen beiden Verfahren finden Sie in Kapitel 6 »Ausdrücke« des *Programmierhandbuches*.

Zusatzinformationen zur Fehlersuche

Syntax:	{$D+} oder {$D-} bzw. /$D+ oder /$D-
Standard:	{$D+}
Typ:	global
Menü-Äquivalent:	*Options/Compiler/Debug information*

Dieser Schalter legt fest, ob der Compiler zusätzliche Informationen zur Fehlersuche erzeugt oder nicht.

Zusatzinformationen zur Fehlersuche bestehen aus einer Tabelle mit Zeilennummern für jede Routine, über die die Verbindung zwischen den Adressen der einzelnen Befehle und dem Quelltext hergestellt werden kann. Turbo Pascal verwendet diese Informationen nicht nur zur Lokalisierung von Laufzeitfehlern, sondern auch für sämtliche Operationen des integrierten Debuggers (schrittweise Ausführung, Abbruchpunkte, »Return-Stack« usw).

Bei der Compilierung von Units werden die Zusatzinformationen zusammen mit Code und Daten direkt in der entsprechenden .TPU-Datei gespeichert. Bei der Compilierung von Programmen wird zwischen zwei Möglichkeiten unterschieden:

Compilierung in den Hauptspeicher: Turbo Pascal reserviert einen Speicherbereich für die Zusatzinformationen. Wenn das Programm danach gestartet wird und mit einem Laufzeitfehler abbricht, geschieht die Zuordnung zur entsprechenden Stelle im Quelltext automatisch.

Compilierung als .EXE-Datei auf die Diskette: hier werden die zusätzlichen Informationen (analog zu Units) als Teil .EXE-Datei gespeichert.

Die Compilierung im Modus {$D+} vergrößert .TPU- und .EXE-Dateien bzw. vergrößert den Platzbedarf bei einer Compilierung im Hauptspeicher – Größe und Arbeitsgeschwindigkeit eines fertigen Programms bleiben aber unbeeinflußt.

Hinweise: Im Modus {$D+} läßt sich über den Schalter {$L} zusätzlich festlegen, ob Daten über lokale Symbole zusammen mit den »Zeilennummern« gespeichert werden oder nicht. Für die Fehlersuche mit dem *Turbo Debugger* ist kein Compiler-Befehl im Quelltext definiert: In der integrierten Entwicklungsumgebung müssen hier die Wahlpunkte *Compile/Destination* und *Options/Debugger/Standalone* auf *Disk* bzw. auf *On* gesetzt sein; bei der Arbeit mit der Kommandozeilen-Version des Compilers erfüllt der Parameter /V dieselbe Funktion.

/$E – Emulator

Syntax:	{$E+} oder {$E-} bzw. /$E+ oder /$E-
Standard:	{$E+}
Typ:	global
Menü-Äquivalent:	Options/Compiler/Emulation

Dieser Schalter legt fest, welche Routinen im Modus {$N+} zur Steuerung des Coprozessors in ein Programm eingebunden werden. Der vom Compiler erzeugte Code wird dadurch nicht beeinflußt – folglich spielt der Stand von {$E} bei der Compilierung von Units keine Rolle.

Im Modus {$N+,E+} nimmt Turbo Pascal den kompletten Emulator in die .EXE-Datei auf – derartige Programme sind auf Systemen mit und ohne Coprozessor lauffähig. Im Modus {{$N+,$E-} werden nur einige Routinen zur direkten Ansteuerung des 8087 in das Programm eingebunden, das dadurch um einige Kilobyte kürzer wird – sich aber nur auf Systemen verwenden läßt, die tatsächlich mit einem Coprozessor ausgerüstet sind.

Hinweis: Wenn ein Programm zusammen mit sämtlichen von ihm benutzten Units im Modus {SN-} compiliert wurde, ist weder die 8087-Bibliothek noch der Emulator erforderlich – der Schalter {$E} wird in diesem Fall ignoriert. Weitere Details finden Sie in Kapitel 13 »Der Coprozessor 8087« des *Programmierhandbuches*.

Erzwingen von *far*-Aufrufen

Syntax:	{$F+} oder {$F-} bzw. /$F+ oder /$F-
Standard:	{$F-}
Typ:	lokal
Menü-Äquivalent:	*Options/Compiler/Force far calls*

Dieser Schalter legt fest, ob der Compiler bei nachfolgenden Prozeduren und Funktionen selbst über die Art des Aufrufs entscheidet oder ob er Aufrufe und Rücksprünge grundsätzlich als *far* codiert.

Im Modus {$F-}werden Prozeduren und Funktionen nur dann als *far* aufgerufen, wenn sie im Interface-Teil einer Unit erscheinen und damit auch anderen Modulen zur Verfügung stehen, ansonsten geschieht die Codierung als *near*. Im

Modus {$F+} wird (außer für lokale Prozeduren und Funktionen) immer mit *far*-Aufrufen und entsprechenden Rücksprungbefehlen gearbeitet.

Um die Codierung einer Prozedur oder Funktion als *far* zu erzwingen, reicht es, nur den Kopf mit {$F+} und {$F-}einzufassen, also

```
{$F+} function Sehr_weit_weg(<Parameter>):<Ergebnis>; {$F-}
```

Der Compiler paßt daraufhin die Art der Zugriffe auf übergebene Parameter sowie die Codierung des Rücksprungbefehls entsprechend an und sorgt dafür, daß jeder Aufruf dieser Routine mit einem CALL FAR erfolgt. Kapitel 14 »Interne Details« des *Programmierhandbuches* enthält weitere Details.

Hinweise: Jedem Programm oder jeder Unit, die mit Overlays arbeiten, sollte der Schalter {$F+} vorangestellt werden; Routinen, die über prozedurale Parameter aufgerufen werden, müssen ebenfalls als *far* codiert sein. Weitere Details zu diesen Themen finden Sie in Kapitel 3 »Units, Overlays und andere Geheimnisse« und Kapitel 4 »Programmierung für Fortgeschrittene« des *Benutzerhandbuches* sowie in Kapitel 8 »Prozeduren und Funktionen« des *Programmierhandbuches.*

Erzeugung von 80286-Code

Syntax:	{$G+} oder {$G-} bzw. /$G+ oder /$G-
Standard:	{$G-}
Typ:	lokal
Menü-Äquivalent:	*Options/Compiler/286 instructions*

Dieser Schalter legt fest, ob der Compiler Code für den 80286-Prozessor erzeugt oder nicht. Im Modus {$G-} werden ausschließlich 8086-Befehle erzeugt. Solchermaßen compilierte Programme sind auf allen Rechnern mit Prozessoren der 80x86-Familie lauffähig.

Im Modus {$G+} hingegen verwendet der Compiler die zusätzlichen Befehle des 80286-Prozessors zur Erhöhung der Effizienz des Codes. Folgende 80286-Befehle sind im Modus {$G+} verfügbar: ENTER, LEAVE, PUSH *konstante*, erweitertes IMUL sowie die erweiterten SHL und SHR. Programme, die im Modus {$G+} compiliert wurden, sind auf Rechnern mit 8088- bzw. 8086-Prozessoren nicht mehr lauffähig.

Automatische Prüfung von Ein-/Ausgaben (I/O)

Syntax:	{$I+} oder {$I-} bzw. /$I+ oder /$I-
Standard:	{$I+}
Typ:	lokal
Menü-Äquivalent:	*Options/Compiler/I/O checking*

Dieser Schalter legt fest, ob Ein- und Ausgaben zur Laufzeit des Programms automatisch auf Fehler überprüft werden.

Im Modus {$I+}erzeugt der Compiler nach jedem Ein-/Ausgabebefehl automatisch zusätzlichen Prüfcode, der das Programm im Falle eines Fehlers mit einer entsprechenden Meldung abbricht. Im Modus {$I-}muß die Fehlerprüfung durch das Programm über Aufrufe der Funktion *IOResult* erfolgen (siehe Kapitel 4 »Die sieben Elemente der Programmierung« des *Benutzerhandbuches*).

/$L – Lokale Symbole

Syntax:	{$L+} oder {$L-} bzw. /$L+ oder /$L-
Standard:	{$L+}
Typ:	global
Menü-Äquivalent:	*Options/Compiler/Local symbols*

Dieser Schalter legt fest, ob Turbo Pascal neben den über {$D} (Debug information) erzeugten Zeilennummern zusätzlich Informationen über »lokale« Symbole in ein Modul aufnimmt oder nicht.

Im Modus {$L+} berücksichtigt der Compiler nicht nur die Namen und Datentypen aller lokalen Variablen und Konstanten des jeweiligen Moduls – bei der Compilierung von Units werden auch die Routinen (zusammen mit ihren lokalen Symbolen) verzeichnet, die nur im Implementationsteil deklariert sind.

Im Modus {$L-}werden »lokale« Bezeichner vollständig ausgeschlossen.

Hinweis: Bei einer Compilierung im Modus {$D-} – also ohne das Erzeugen von Informationen zur Fehlersuche – bleibt der Schalter {$L} wirkungslos.

Numerische Operationen

Syntax:	{$N+} oder {SN-} bzw. /$N+ oder /$N-
Standard:	{$N-}
Typ:	global
Menü-Äquivalent:	*Options/Compiler/8087/80287*

Dieser Schalter legt fest, ob Operationen mit Realzahlen durch Routinen der Laufzeitbibliothek oder über die direkte Ansteuerung eines numerischen Coprozessors stattfinden.

Im Modus {SN-} (Software) steht nur der Datentyp *Real* zur Verfügung, alle Operationen mit diesem Typ geschehen über Aufrufe der Laufzeitbibliothek.

Im Modus {$N+} (8087/80287) bietet Turbo Pascal vier zusätzliche Real-Datentypen und setzt einen numerischen Coprozessor (8087, 80287 oder 80387) bzw. die Einbindung des Emulators über den Compiler-Schalter {$E+} voraus (siehe Kapitel 13 »Der Coprozessor 8087« des *Programmierhandbuches*).

/$O – Overlay-Prüfung

Syntax:	{$O+} oder {$O-} bzw. /$O+ oder /$O-
Standard:	{$O-}
Typ:	global
Menü-Äquivalent:	*Options/Compiler/Overlays allowed*

Dieser Schalter legt fest, ob der Compiler Units »Overlay-fähig« macht oder nicht.

Im Modus {$O+} führt Turbo Pascal einige zusätzliche Prüfungen für die Übergabe von String- und **set**-Konstanten zwischen Overlay-Routinen durch und stellt (nötigenfalls durch mehrfache Speicherung auf dem Stack) sicher, daß sich dabei keine der in Kapitel 3 »Units, Overlays und andere Geheimnisse« des *Benutzerhandbuches* besprochenen Probleme ergeben.

Hinweis: Eine mit {$O+} übersetzte Unit wird nicht automatisch zum Overlay – Turbo Pascal sorgt hier nur dafür, daß es als Overlay verwendet werden *kann*.

Bereichs-Überprüfung

Syntax:	`{$R+}` oder `{$R-}` bzw. `/$R+` oder `/$R-`
Standard:	`{$R-}`
Typ:	lokal
Menü-Äquivalent:	*Options/Compiler/Range checking*

Dieser Schalter legt fest, ob der Compiler bei der Indizierung von Strings und Arrays sowie bei Zuordnungen zu skalaren Typen und Unterbereichen zusätzlichen Prüfcode erzeugt.

Im Modus `{$R+}` wird vor jeder Indizierung und jeder Zuordnung geprüft, ob sie sich innerhalb der zulässigen Grenzen bewegt – falls nicht, bricht die Programmausführung mit einer entsprechenden Meldung ab.

Zusätzlich werden im Modus `{$R+}` sämtliche Aufrufe virtueller Methoden auf den Initialisierungs-Status der aufrufenden Objektinstanz hin überprüft. Wurde die aufrufende Instanz noch nicht durch ihren Konstruktor initialisiert, wird ebenfalls dieser Laufzeitfehler ausgelöst.

Die zusätzliche Bereichs-Überprüfung vergrößert ebenso wie auch die Überprüfung der Aufrufe virtueller Methoden den Umfang der .EXE-Datei und setzt die Ausführungsgeschwindigkeit erheblich herab. Sie sollten den Schalter `{$R+}` deshalb ausschließlich in der Testphase aktivieren.

Stack-Prüfung

Syntax:	`{$S+}` oder `{$S-}` bzw. `/$S+` oder `/$S-`
Standard:	`{$S+}`
Typ:	lokal
Menü-Äquivalent:	*Options/Compiler/Stack checking*

Dieser Schalter legt fest, ob der Compiler zusätzlichen Prüfcode vor Belegungen des Stacks erzeugt.

Im Modus `{$S+}` wird vor jedem Prozedur- und Funktionsaufruf geprüft, ob noch genügend Platz zur Speicherung von Rücksprungadressen, lokalen Variablen usw. auf dem Stack vorhanden ist – falls nicht, bricht das Programm mit einer entsprechenden Meldung ab. Auch wenn diese Prüfung den Umfang des Programms erhöht und die Ausführungsgeschwindigkeit herabsetzt, sollte sie wirklich erst dann abgeschaltet werden, wenn das Programm völlig fehlerfrei ist: Stack-Überläufe haben normalerweise einen Systemabsturz zur Folge und sind äußerst schwer zu lokalisieren.

Überprüfung von var-Strings

Syntax:	`{$V+}` oder `{$V-}` bzw. `/$V+` oder `/$V-`
Standard:	`{$V+}`
Typ:	lokal
Menü-Äquivalent:	*Options/Compiler/strict var-strings*

Dieser Schalter legt fest, welche Prüfungen der Compiler bei der Übergabe von Strings als **var**-Parameter vornimmt.

Im Modus `{$V+}` (Strict) setzt der Compiler voraus, daß formale und aktuelle String-Parameter einen *identischen* Typ haben, und weist jede Verletzung dieser Regel als Fehler aus.

Im Modus {$V-} (Relaxed) kann jeder String als **var**-Parameter übergeben werden – selbst dann, wenn die Länge des formalen und die des aktuellen Strings nicht gleich sind.

Erweiterte Syntax

Syntax:	{$X+} oder {$x-} bzw. /$X+ oder /$X-
Standard:	{$X-}
Typ:	global
Menü-Äquivalent:	nicht definiert

Dieser Schalter legt fest, ob die erweiterte Syntax von Turbo Pascal eingesetzt werden kann oder nicht.

Funktionsanweisungen: Im Modus {$X+} ist es möglich, Funktionsaufrufe als Anweisungen zu verwenden, d.h. eine Funktion aufzurufen, ihr Ergebnis jedoch zu verwerfen. Eigentlich zeichnet sich eine Funktion gerade dadurch aus, daß ihre Berechnungen immer zu einem Ergebnis führen, und daher erscheint es nur wenig sinnvoll, ein Funktionsergebnis zu verwerfen. In manchen Situationen ist bei einem Funktionsaufruf jedoch nicht das Funktionsergebnis relevant, sondern vielmehr die Ausführung verschiedener Operationen in Abhängigkeit der Funktionsparameter. Der Schalter {$X+} ermöglicht für solche Fälle die Handhabung einer Funktion entsprechend einer Prozedur.

Hinweis: Der Schalter {X+} hat keine Auswirkung auf die Handhabung der Funktionen der Unit *System*.

Optionales Zeigersymbol: Das Konstrukt

```
Zeiger^.Feld
```

kann abgekürzt werden in

```
Zeiger.Feld
```

Mit anderen Worten, die Zeichensequenz "^." kann mit "." abgekürzt werden. Diese Neuerung stellt ihren Nutzen besonders bei der gemischten Verwendung statischer und dynamischer Objekte unter Beweis, da die Syntax für den Zugriff auf ein Feld bzw. eine Methode immer die gleiche bleibt – gleichgültig, ob das Objekt statischer oder dynamischer Natur ist. Mit der Standardeinstellung {$X-} verursacht der Versuch, die Syntax-Erweiterungen anzuwenden, einen Fehler.

Parameter

Include-Datei

Syntax:	{$I *Dateiname*}
Typ:	lokal
Menü-Äquivalent:	*Options/Directories/Include directories*

Dieser Befehl weist den Compiler an, die durch *Dateiname* bezeichnete Datei einzulesen. Der Compiler verhält sich dabei im Prinzip so, als ob der gelesene

Text anstelle des {$I}Befehls stünde – die eingelesene Datei muß allerdings einen abgeschlossenen Programmblock enthalten (d.h. vollständige Prozeduren, Funktionen und/oder ein komplettes Hauptprogramm inklusive **begin** und **end**).

Wenn *Dateiname* kein Suffix enthält, wird automatisch .PAS angenommen. Wenn *Dateiname* keinen Suchweg angibt, dann sucht Turbo Pascal zuerst im momentan gesetzten Verzeichnis und danach in sämtlichen Verzeichnissen, die via *Options/Directories/Include directories* bzw. durch den Kommandozeilen-Parameter */I* angegeben sind.

Turbo Pascal erlaubt die Verschachtelung von Include-Dateien bis zu 15 Ebenen. Eine via {$I} eingefügte Datei kann also ihrerseits weitere Dateien einfügen, die wiederum {$I}Befehle enthalten usw.

Einbinden von Object-Dateien

Syntax:	{$L *Dateiname*}
Typ:	lokal
Menü-Äquivalent:	*Options/Directories/Object directories*

Dieser Befehl weist den Linker an, die durch *Dateiname* bezeichnete Datei mit in das Programm aufzunehmen. Die Datei muß im Intel-Object-Format vorliegen (muß also mit einem Assembler erzeugt sein). Wenn kein Suffix angegeben ist, dann hängt Turbo Pascal automatisch die Endung .OBJ an *Dateiname* an.

Bezeichner der Object-Datei, auf die von Pascal aus zugegriffen wird, müssen innerhalb der Object-Datei als PUBLIC und innerhalb des Pascal-Quelltextes als **external** deklariert sein (siehe Kapitel 14 »Interne Details« des *Programmierhandbuches*).

Wenn *Dateiname* keinen Suchweg angibt, dann sucht Turbo Pascal zuerst im momentan gesetzten Verzeichnis, und danach in sämtlichen Verzeichnissen, die über *O/D/Object directories* bzw. den Kommandozeilen-Parameter /O angegeben wurden.

Speicherbelegung

Syntax:	{$M Stackgröße , HeapMin , HeapMax }
Standard:	{$M 16384,0,655360}
Typ:	global
Menü-Äquivalent:	*Options/Memory sizes*

Diese Parameter legen fest, wieviel Platz Turbo Pascal für den Stack (Stackgröße«, Menüpunkt ÖMIStack size) reserviert und wieviel Platz minimal (»HeapMin«, Menüpunkt *Low heap limit*) bzw. maximal (»HeapMax«, Menüpunkt *High heap limit*) für den Heap-Bereich belegt werden sollen.

Zulässige Werte für die Stackgröße liegen im Bereich von 1024 bis 65520; für *HeapMin* und *HeapMax* im Bereich von 0..655360 (640 KByte), wobei *HeapMin* kleiner oder gleich *HeapMax* sein muß. Kapitel 4 »Variablen« und Kapitel 14 »Interne Details« des *Programmierhandbuches* enthalten weitere Einzelheiten hierzu.

Hinweis: Der Befehl {$M} hat bei der Compilierung einer Unit keine Wirkung.

Overlay-Deklaration

Syntax:	`{$O Unitname}`
Typ:	lokal
Menü-Äquivalent:	nicht definiert

Dieser Befehl deklariert eine Unit als Overlay: er teilt dem Linker mit, daß der entsprechende Code nicht direkt im Programm, sondern in einer Datei mit dem Suffix .OVR gespeichert werden soll.

Auf den erzeugten Code hat der Parameter `{$O}` keine Wirkung – folglich bleibt er bei der Compilierung von Units wirkungslos.

Die als *Unitname* angegebene Unit muß zuvor mit **uses** aufgeführt und im Modus `{$O+}` compiliert worden sein. Weitere Details finden Sie in Kapitel 3 »Units,Overlays und andere Geheimnisse« des *Benutzerhandbuches* .

Bedingte Compilierung

Turbo Pascal stellt zwei Konstrukte zur Verfügung, mit denen Teile eines Quelltextes von der Compilierung ausgeschlossen bzw. in die Compilierung einbezogen werden können. Das erste Konstrukt hat die Form

```
{$IF xxxx} ... {$ENDIF}
```

Der mit `{$IF}`und `{$ENDIF}` eingeschlossene Text wird nur dann vom Compiler bearbeitet, wenn die mit *xxxx* angegebene Bedingung zutrifft und ansonsten ignoriert.

Das zweite Konstrukt stellt eine Erweiterung des ersten dar und hat die Form

```
{$IF xxxx} ... {$ELSE} ... {$ENDIF}
```

Wenn die durch *xxxx* angegebene Bedingung zutrifft, wird der mit `{$IF}`und `{$ELSE}` eingeschlossene Text vom Compiler bearbeitet (und der zwischen `{$ELSE}` und `{$ENDIF}` befindliche Text ignoriert). Trifft die durch *xxxx* angegebene Bedingung nicht zu, dann ignoriert der Compiler den Text zwischen *{$IF}*/*{$ELSE}* und bearbeitet den durch `{$ELSE}` und `{$ENDIF}` eingeschlossenen Teil.

Mit diesen beiden Befehlen zur bedingten Compilierung lassen sich Programme schreiben, die durch die Veränderung eines einzigen Symbols an verschiedene Gerätekonfigurationen angepaßt oder für verschiedene Zwecke verwendet werden können. Einige Beispiele dazu:

```
{$IFDEF Debug}
  Writeln('X = ', X);
{$ENDIF}

{$IFDEF CPU87}
  {$N+}
  type
    Real = Double;
```

```
{$ELSE}
  {$N-}
  type
    Single = Real;
    Double = Real;
    Extended = Real;
    Comp = Real;
{$ENDIF}
```

Befehle zur bedingten Compilierung lassen sich bis zu einer Tiefe von 16 Ebenen verschachteln. Über bedingte Befehle lassen sich nicht nur Typen definieren, Compiler-Schalter setzen oder Units aufnehmen – auch Include-Dateien können darüber einbezogen oder ausgeschlossen werden. Für Include-Dateien gibt es aber eine Voraussetzung: Jede mit `{$IF..}` begonnene Verschachtelungsebene muß innerhalb derselben Datei wieder abschlossen sein, d.h. jede Datei muß genausoviele `{$IF..}` wie `{$ENDIF}`-Befehle enthalten.

Symbole zur bedingten Compilierung

Die Auswertung einer Bedingung *xxxx* für den Befehl `{$IFDEF}` ist recht einfach und im Stil boolscher Variablen gehalten: *xxxx* steht in jedem Fall für ein *Symbol*, das entweder definiert ist (in diesem Fall trifft die Bedingung zu) oder nicht (die `{$IF..}`-Bedingung ergibt *False*). Zur Definition und Löschung (Undefinition) derartiger Symbole gibt es zwei weitere Compiler-Befehle:

```
{$DEFINE Symbolname}     { definiert ein Symbol }
{$UNDEF Symbolname}      { löscht ein Symbol }
```

Symbole zur bedingten Compilierung lassen sich zusätzlich über den Menüpunkt *O/C/Conditional defines* bzw. über den Kommandozeilen-Parameter `/D` erzeugen. Für ihre Namen gelten dieselben Regeln wie für Bezeichner von Turbo Pascal: das erste Zeichen muß ein Buchstabe sein, darauf können beliebige Kombinationen von Buchstaben, Ziffern und Unterstrichen (_) folgen. Zwischen Groß- und Kleinbuchstaben wird nicht unterschieden, die deutschen Umlaute und das »ß« sind nicht zulässig. Die Länge eines Namens ist beliebig, die ersten 63 Zeichen sind signifikant.

Hinweis: Symbole zur bedingten Compilierung und Pascal-Bezeichner sind vollständig voneinander getrennt. Dadurch können Probleme mit doppelt vergebenen Namen erst gar nicht entstehen, ein Zugriff auf bedingte Symbole von einem Pascal-Programm oder umgekehrt ist allerdings auch nicht möglich. Im folgenden Beispiel erhalten Sie weder die Fehlermeldung »doppelter Bezeichner« noch wird der *Writeln*-Befehl vom Compiler bearbeitet:

```
{$UNDEF Debug}   { Symbol zur bedingten Compilierung }

const
  Debug = True;   { Pascal-Variable }
begin
```

```
  {$IFDEF Debug}
    Writeln('Debug ist definiert...');
  {$ENDIF}
end.
```

Aus demselben Grund endet die Compilierung des nächsten Beispiels mit der Meldung »unbekannter Bezeichner«:

```
var             { diese »var-Sektion« ist erlaubt - sie hat aber }
{$DEFINE Debug}  { nichts mit einer Pascal-Variablen zu tun! }
begin
   if Debug then Writeln('Debug ist definiert...');
end.
```

Die folgende Liste gibt die Symbole wieder, die Turbo Pascal zu Beginn jeder Compilierung definiert. Diese Symbole können zusammen mit `{$IF}`verwendet (oder auch mit `{$UNDEF}` gelöscht) werden.

`VER60`: steht für die Version 6.0 des Compilers. Die Version 4.0 von Turbo Pascal definiert ein Symbol namens `VER40`; folgende Versionen werden ebenfalls eigene Symbole definieren (`VER50` für die Version 5.0 usw).

`MSDOS`: steht dafür, daß der Compiler das Betriebssystem MS-DOS (oder PC-DOS) voraussetzt. Versionen von Turbo Pascal, die unter anderen Betriebssystemen arbeiten, werden ein entsprechendes Symbol anderen Namens definieren.

`CPU86`: steht dafür, daß der Compiler einen Prozessor der Familie 80x86 voraussetzt. Versionen von Turbo Pascal, die für andere Prozessorfamilien geschrieben sind, werden ein entsprechendes Symbol anderen Namens definieren.

`CPU87`: ist das einzige Symbol, das nicht immer definiert ist: Es steht für das Vorhandensein eines numerischen Coprozessors (8087) zum Zeitpunkt der Compilierung. Über dieses Symbol läßt sich das Setzen des Schalters `{$E}` automatisieren:

```
{$IFDEF CPU87} {$N+,E-} {$ELSE} {$N+,E+} {$ENDIF}
```

Zusätzlich könnten hier die Datentypen *Single*, *Double* usw. entsprechend definiert werden – siehe Anfang dieses Abschnitts.

Der bedingte Compiler-Befehl DEFINE

Syntax: `{$DEFINE Symbolname}`

Definiert das durch *Symbolname* angegebene Symbol, das vom Zeitpunkt seiner Definition bis zum Ende der Compilierung oder einem entsprechenden `{$UNDEF}` -Befehl erhalten bleibt. Wenn *Symbolname* vor diesem Befehl bereits definiert war, dann hat `{$DEFINE}` keine Wirkung.

Der bedingte Compiler-Befehl UNDEF

Syntax: `{$UNDEF Symbolname}`

Löscht (undefiniert) das durch *Symbolname* bezeichnete Symbol bis zum Ende der Compilierung bzw. bis zu seiner erneuten Definition via `{$DEFINE}` . Wenn

Symbolname vor diesem Befehl überhaupt nicht definiert war, dann hat {*$UNDEF*} keine Wirkung.

Der bedingte Compiler-Befehl IFDEF

Syntax: `{$IFDEF Symbolname }`

Bewirkt die Compilierung des folgenden Textes, wenn das durch *Symbolname* bezeichnete Symbol definiert ist. Ansonsten wird der folgende Text bis zum nächsten $ENDIF oder $ELSE ignoriert.

Der bedingte Compiler-Befehl IFNDEF

Syntax: `{$IFNDEF Symbolname }`

Das logische Gegenstück zu `{$IFDEF}` – nachfolgender Text wird nur compiliert, wenn das durch *Symbolname* bezeichnete Symbol *nicht* definiert ist.

Der bedingte Compiler-Befehl IFOPT

Syntax: `{$IFOPT Compiler-Schalter }`

Bestimmt die Compilierung nachfolgenden Textes abhängig vom Stand eines angegebenen Compiler-Schalters. *Compiler-Schalter* steht hier für den Namen eines Schalters, gefolgt von einem Plus- oder Minuszeichen. Wenn die Angabe mit dem tatsächlichen Stand des Schalters übereinstimmt, gilt die Bedingung als erfüllt. Die **type**-Definition im folgenden Beispiel wird also nur dann compiliert, wenn der Schalter `$N` auf `{$N+}` gesetzt ist:

```
{$IFOPT N+}
  type  Real = Extended;
{$ENDIF}
```

Der bedingte Compiler-Befehl ELSE

Syntax: `{$ELSE}`

Leitet den **else**-Zweig eines mit `{$IFDEF}`, `{$IFNDEF}` oder `{$IFOPT}` begonnenen Konstrukts ein und bezieht sich immer auf das letzte noch nicht abgeschlossene `{$IF...}`

Der bedingte Compiler-Befehl ENDIF

Syntax: `{$ENDIF}`

Schließt ein mit `{$IFDEF}`, `{$IFNDEF}` oder `{$IFOPT}` begonnenes Konstrukt ab und bezieht sich immer auf das letzte noch nicht abgeschlossene `{$IF...}`

3. Fehlermeldungen und -codes

Turbo Pascal unterscheidet (wie jeder andere Compiler auch) zwischen zwei Arten von Fehlern, die eigentlich recht wenig miteinander zu tun haben:

- Fehler während der Compilierung werden durch *syntaktisch* unzulässige Konstrukte wie fehlende oder überzählige Elemente, falschgeschriebene Schlüsselwörter und/oder Variablennamen sowie durch Beschränkungen des Compilers und des Systems (mehr als 64 KByte Code in einem Segment, zuviele Units, mehr als 64 KByte globale Variablen usw.) hervorgerufen. Die dazugehörigen Meldungen werden vom Compiler ausgegeben.

- Fehler während des Programmlaufs – auch allgemein als Laufzeitfehler bezeichnet – entstehen beispielsweise beim Versuch, eine Variable über ihren zulässigen Wertebereich hinaus zu erhöhen oder Zeichen zur Tastatur zu schreiben. Da der Compiler keine Annahmen über den Verlauf des Programms macht – er kann beispielsweise nicht bestimmen, wie oft eine mit **while** aufgebaute Schleife wiederholt wird – treten diese *semantischen* Fehler erst zur Laufzeit des Programms in Erscheinung. Die Codes dieser Fehler werden durch Routinen der Laufzeitbibliothek definiert – Meldungen im Klartext erhalten Sie nur dann, wenn das Programm innerhalb der integrierten Entwicklungsumgebung gestartet wurde.

Fehlermeldungen des Compilers

Wenn der Compiler einen Fehler erkennt, gibt er nicht nur eine entsprechende Meldung aus, sondern auch noch soviele Zusatzinformationen wie möglich in Form von Bezeichnern oder Dateinamen. Ein Beispiel:

```
Error 15:File not found (WINDOW.TPU)
```

Innerhalb der integrierten Entwicklungsumgebung wird die Datei, in der der Fehler erkannt wurde, automatisch in den Editor geladen und der Cursor entsprechend positioniert. Der Text der Fehlermeldung erscheint in der Statuszeile des Fensters *Edit* und verschwindet beim ersten Tastendruck, um der normalen Anzeige des Editors Platz zu machen (mit Ctrl Q W läßt sie sich jederzeit erneut auf den Bildschirm bringen).

Der Punkt, an dem ein Fehler vom Compiler erkannt wird, liegt in verschiedenen Fällen einige Zeilen hinter der tatsächlichen Fehlerstelle im Quelltext – das folgende Beispiel zeigt, warum:

```
begin
if x < 4 then begin
   if y < 5 then begin
        ....
    end;
     ....
     ... <- irgendwo hier fehlt ein "end;"
     ...
end.      <- Fehler erkannt
```

Hier erkennt der Compiler erst durch das Ende des Hauptprogramms, daß der Programmierer irgendwo ein **end** (für `if x < 4 then begin`) vergessen hat – wo dieses **end** eingefügt werden muß, kann er natürlich nicht bestimmen.

Ähnliches ist bei der Auswertung sehr komplexer Ausdrücke der Fall: der Compiler erkennt unter Umständen erst bei der Codierung einer Zuweisung, daß der Ausdruck einen unzulässigen Datentyp ergeben hat. Derartige Konstrukte sind der einzige Fall, in dem der Cursor *vor* die eigentliche Fehlerstelle gesetzt wird.

1 Out of memory

(Platz im Hauptspeicher reicht nicht aus)

Für diesen Fehler gibt es eine Reihe von Gründen und Möglichkeiten zur Behebung:

- Innerhalb der integrierten Entwicklungsumgebung: Versuchen Sie es mit der Compilierung auf die Diskette (*Compile/Destination..Disk*).

- Verwenden Sie *Options/Linker/Link buffer...Disk*, damit der Link-Puffer auf Disk angelegt wird oder die Option `/L` in der Kommandozeilenversion.

- Entfernen Sie speicherresidente Zusatzprogramme (wie SideKick oder SuperKey).

- Wenn sie mit der integrierten Entwicklungsumgebung arbeiten und alle diese Maßnahmen nichts helfen: Versuchen Sie es mit der Kommandozeilen-Version (die rund 50 KByte weniger Speicherplatz belegt).

- Die Unit-Bibliothek (TURBO.TPL) wird von beiden Compilerversionen beim Start eingelesen und belegt ebenfalls Platz im Hauptspeicher. Falls Sie hier sehr viele (und sehr große) eigene Units aufgenommen haben, sollten Sie diese Units (mit TPUMOVER) auf einzelne Dateien verteilen.

Der vorletzte (und komplizierteste) Ausweg: Verteilen Sie Ihr Programm auf mehrere Units und compilieren Sie diese Units separat. Diese Units dürfen *nicht* in TURBO.TPL aufgenommen werden – s. oben.

Wenn Ihr Rechner über mindestens 1 MByte freies Extended Memory verfügt, können Sie den Compiler TPCX verwenden, der im Protected Mode läuft.

2 Identifier expected
(Bezeichner erwartet)
Der Compiler erwartet an dieser Stelle einen Bezeichner bzw. Ihre Angabe ist aus irgendeinem Grund nicht als Bezeichner akzeptabel. Diesen Fehler erhalten Sie auch beim Versuch, ein reserviertes Wort zu redeklarieren.

3 Unknown identifier
(unbekannter Bezeichner)
Der angegebene Bezeichner ist dem Compiler unbekannt (entweder haben Sie die Deklaration vergessen oder sich irgendwo vertippt).

4 Duplicate identifier
(doppelter Bezeichner)
Der angegebene Bezeichner ist innerhalb des momentanen Blocks bereits vergeben.

5 Syntax error
(Syntaxfehler)
Der Compiler hat ein unzulässiges Zeichen innerhalb des Quelltextes entdeckt. (Möglicher Grund: Ein Steuerzeichen für den Drucker o.ä. außerhalb eines Kommentars oder außerhalb eines mit Hochkommata eingeschlossenen Strings).

6 Error in real constant
(fehlerhafte Angabe einer Real-Konstanten)
Die Syntax von Real-Konstanten ist in Kapitel 1 »Sprachelemente und Konstanten« des *Programmierhandbuches* beschrieben.

7 Error in integer constant
(fehlerhafte Angabe einer Integer-Konstanten)
Ganzzahlige Werte müssen mit einem Dezimalpunkt und einer folgenden Null angegeben werden, wenn sie außerhalb des Wertebereichs für Integertypen liegen (z.B. 8844349587.0). Die Syntax von Integer-Konstanten ist in Kapitel 1 »Sprachelemente und Konstanten« des *Programmierhandbuches* beschrieben.

8 String constant exceeds line
(String-Konstante geht über das Zeilenende hinaus)
String-Konstanten müssen grundsätzlich in der Zeile des Quelltextes abgeschlossen werden, in der sie auch begonnen haben. Wahrscheinlich haben Sie ein abschließendes Hochkomma vergessen.

9 Too many nested files
(Include-Verschachtelung zu tief)
Verschachtelungen über den Compiler-Befehl {*$I <Dateiname>*} sind maximal bis zu einer Tiefe von 15 Ebenen erlaubt.

10 Unexpected end of file
(unerwartetes Quelltext-Ende)

Für diese Fehlermeldung gibt es drei wahrscheinliche Gründe:

- Ihre Quelltext-Datei endet vor dem abschließenden **end.** des Hauptprogramms, Sie haben das Hauptprogramm versehentlich mit einem Semikolon (**end;**) abgeschlossen oder der Text enthält allgemein eine nicht ausgeglichene Anzahl von **begin** und **end**.
- Eine Include-Datei endet inmitten eines Anweisungsteils. (Die Aufteilung eines Prozedur- oder Funktionsblocks auf mehrere Dateien ist nicht möglich).
- Sie haben vergessen, einen Kommentar abzuschließen.

11 Line too long
(Programmzeile hat mehr als 126 Zeichen)

12 Type identifier expected
(Typ-Bezeichner erwartet)

13 Too many open files
(zuviele Dateien auf einmal offen)
Das System reserviert nicht genügend Speicherplatz für gleichzeitig offene Dateien. Prüfen Sie den Parameter FILES in der Datei CONFIG.SYS und erhöhen Sie ihn gegebenenfalls (FILES=20). Danach ist ein Neustart des Computers erforderlich, um die Änderungen wirksam werden zu lassen.

14 Invalid file name
(ungültiger Dateiname)
Der in einem Compiler-Befehl angegebene Dateiname und/oder Suchweg ist ungültig bzw. der Suchweg existiert nicht.

15 File not found
(Datei nicht gefunden)
Die in einem Compiler-Befehl angegebene Datei läßt sich weder im momentan gesetzten Verzeichnis noch über die Suchwege (*Options/*) für den entsprechenden Dateityp finden. Diesen Fehler erhalten Sie auch dann, wenn der Compiler eine mit **uses** angebene Unit nicht findet.

16 Disk full
(Diskette/Festplatte voll)
Löschen Sie nicht mehr benötigte Dateien (*.BAK).

17 Invalid compiler directive
(Compiler-Befehl ungültig)
Entweder ist der nach {$ angegebene Buchstabe nicht als Compiler-Befehl definiert, ein angegebener Parameter ungültig oder Sie haben einen Befehl im Rumpf des Programms angegeben, der globale Wirkung hat und deshalb nur vor der ersten Prozedur- oder Funktionsdeklaration gegeben werden darf.

18 Too many files
(zuviele Dateien innerhalb eines Projekts)
Die interne Dateinamen-Tabelle des Compilers ist voll. Nehmen Sie Include-Dateien direkt in Ihre Quelltexte auf, fassen Sie sämtliche Dateien in einem Verzeichnis zusammen, das als Standard gesetzt wird, und verwenden Sie kürzere Dateinamen. (Turbo Pascal muß jeden Suchweg einzeln speichern – Angaben mit langen Suchwegen wie z.B. {$U C:\TURBO\UNITS\PROJS\SCREEN\PART2.TPU} füllen die Tabelle natürlich wesentlich schneller als einzelne Dateinamen).

19 Undefined type in pointer definition
(Ziel-Typ der Zeigerdefinition ist nicht definiert)
Der bei der Definition eines Zeigers angegebene Datentyp ist dem Compiler nicht bekannt.

20 Variable identifier expected
(Variablen-Bezeichner erwartet)

21 Error in type
(unzulässiges Symbol/Zeichen in einer Typ-Definition)

22 Structure too large
(Datenstruktur ist größer als 65520 Bytes)

23 Set base type out of range
(Mengen-Grundtyp außerhalb des möglichen Wertebereichs)
Der Grundtyp einer Menge muß ein Teilbereich sein, dessen untere Grenze größer/gleich 0 und dessen obere Grenze kleiner/gleich 255 ist, oder ein Aufzählungstyp mit maximal 256 verschiedenen Werten.

24 File components may not be files or objects
[Dateikomponenten dürfen keine Dateien oder Objekte sein.]
Als zusammengesetzten Datentyp einer Datei darf man keinen Objekt- oder Dateityp angeben. Es darf auch kein strukturierter Datentyp mit einer Objekt- oder Dateikomponente angegeben werden.

25 Invalid string length
(Stringlänge außerhalb des Bereichs 1..255)

26 Type mismatch
(Typen nicht miteinander vereinbar)
Dieser Fehler kann einen der folgenden Gründe haben:

Bei Zuweisungen: Der Typ der Variablen und der Ergebnistyp des Ausdrucks sind nicht implizit kompatibel (d.h. der Compiler nimmt eine entsprechende Typ-Umwandlung nicht selbständig vor).

Bei Prozedur- und Funktionsaufrufen: Formaler und aktueller Parameter sind nicht implizit kompatibel.

Bei Indizierungen: Der Ergebnistyp des Ausdrucks und der Indextyp des Arrays sind nicht kompatibel.

Bei Ausdrücken: Die Kombination von Operanden und Operatoren ist nicht miteinander verträglich (Vergleich von *Integer* und *Char* etc).

27 Invalid subrange base type
(Grundtyp des Teilbereichs ist nicht ordinal)

28 Lower bound greater than upper bound
(Untergrenze eines Teilbereichs >Obergrenze)

29 Ordinal type expected
(Ordinaltyp erwartet)
Real-Typen, Strings, strukturierte Typen und Zeiger sind an dieser Stelle nicht verwendbar.

30 Integer constant expected
(Integer-Konstante erwartet)

31 Constant expected
(Konstante erwartet)

32 Integer or real constant expected
(Integer- oder Real-Konstante erwartet)

33 Pointer type identifier expected
(Zeiger-Typ-Bezeichner erwartet)
Der Bezeichner repräsentiert keinen Zeiger-Typ

34 Invalid function result type
(Ergebnistyp der Funktion nicht zulässig)
Gültige Typen von Funktionsergebnissen sind alle einfachen Typen, Strings und Zeiger.

35 Label identifier expected
(Label-Bezeichner erwartet)

36 BEGIN expected
(**begin** erwartet)

37 END expected
(**end** erwartet)

38 Integer expression expected
(Ergebnistyp des Ausdrucks muß *Integer* sein)

39 Ordinal expression expected
(Ergebnistyp des Ausdrucks muß ordinal sein)

40 Boolean expression expected
(Ergebnistyp des Ausdrucks muß *Boolean* sein)

41 Operand types do not match operator
(Operandentypen passen nicht zum Operator)
Der Operator kann auf Operanden dieses Typs nicht angewendet werden (zum Beispiel `'A' div '2'`).

42 Error in expression
(Fehler innerhalb des Ausdrucks)
Der Compiler hat zwar verstanden, daß es sich um einen Ausdruck handelt, kann diesen Ausdruck aber nicht auswerten (z.B. wegen eines vergessenen Operators).

43 Illegal assignment
(Zuordnung nicht erlaubt)
Die wahrscheinlichsten Gründe für diese Fehlermeldung sind:

- Der Versuch, Datei-Variablen oder einer untypisierten Variablen direkt einen Wert zuzuordnen.
- Der Versuch, einer Funktion außerhalb ihres Blocks einen Wert zuzuordnen.

44 Field identifier expected
(Feld-Bezeichner erwartet)
Der angegebene Bezeichner steht nicht für ein Feld der angegebenen Record-Variablen.

45 Object file too large
(Linker: .OBJ-Datei ist größer als 64 KByte)

46 Undefined external
(Linker: **external**-Bezeichner ist nirgendwo definiert)
Ein als **external** deklarierter Bezeichner hat sich in keiner der beteiligten Dateien gefunden. Prüfen Sie, ob alle benötigten Object-Dateien auch über den Compiler-Befehl `{$L <Dateiname>}` angegeben sind, dann Ihre PUBLIC-Deklarationen innerhalb der Assembler-Quelltexte und schließlich auf eventuelle Tippfehler.

47 Invalid object file record
(Linker: .OBJ-Dateiformat unleserlich)
Der Linker kann eine via `{$L <Dateiname>}` angegebene Datei nicht fehlerfrei einbinden. Eine Beschreibung der Voraussetzungen für Maschinenprogramme finden Sie in Kapitel 14 »Interne Details« des *Programmierhandbuches*.

48 Code segment too large

(Codesegment-Grenzen überschritten)

Die Compilierung eines Programms oder einer Unit erzeugt mehr als 65520 Bytes Code. Erstellen Sie eine zusätzliche Unit, in der Sie einige Prozeduren und/oder Funktionen »auslagern«, und nehmen Sie diese Unit mit **uses** in das ursprüngliche Programm (oder die Unit) auf.

49 Data segment too large

(Datensegment-Grenzen überschritten)

Ihr Programm und sämtliche aufgenommenen Units deklarieren zusammen mehr als 64 KByte globaler Daten. Es bleibt Ihnen nichts anderes übrig, als zumindest die größeren Datenstrukturen über Zeiger zu deklarieren und sie über Aufrufe von *New* dynamisch zu erzeugen (d.h. Sie müssen wahrscheinlich größere Teile Ihres Programms umschreiben).

50 DO expected

(**do** erwartet)

51 Invalid PUBLIC definition

(Linker: PUBLIC-Definition inkorrekt)

Diese Fehlermeldung können Sie nur beim Einbinden von Maschinenprogrammen erhalten. Mögliche Gründe sind:

- Ein Bezeichner ist über PUBLIC als global verfügbar definiert worden, der Pascal-Quelltext enthält aber keine entsprechende **external**-Deklaration.
- Ein und derselbe Bezeichner ist in mehreren Object-Dateien als PUBLIC definiert worden.
- Ein als PUBLIC definierter Bezeichner befindet sich außerhalb des CODE-Segments in der .OBJ-Datei.

52 Invalid EXTRN definition

(Linker: EXTRN-Definition inkorrekt)

Diese Fehlermeldung können Sie nur beim Einbinden von Maschinenprogrammen erhalten. Mögliche Gründe sind:

- Ein Bezeichner ist innerhalb des Maschinenprogramms als EXTRN definiert, findet sich aber in keiner Pascal-Datei als global verfügbar (d.h. als eigenständige Prozedur/Funktion innerhalb des Programms oder als Deklaration innerhalb eines Interface-Teils).
- Der Bezeichner ist im Pascal-Programm als **absolute** deklariert.
- Der Bezeichner steht innerhalb des Pascal-Programms für eine als **inline** deklarierte Routine.

53 Too many EXTRN definitions

(Linker: zuviele EXTRN-Definitionen)

Maximal sind 256 EXTRN-Definitionen pro Object-Datei möglich.

54 OF expected
(**of** erwartet)

55 INTERFACE expected
(**interface** erwartet)

56 Invalid relocatable reference
(Zugriff auf relozierbares Symbol inkorrekt)
Diese Fehlermeldung können Sie nur beim Einbinden von Maschinenprogrammen erhalten. Mögliche Gründe sind:

- Die .OBJ-Datei definiert relozierbare Symbole außerhalb des Segments CODE.
- Die .OBJ-Datei enthält Zugriffe auf einzelne Bytes relozierbarer Symbole – beispielsweise die Operatoren HIGH und LOW oder die DB-Direktiven (define byte).
- Ein Operand innerhalb der .OBJ-Datei bezieht sich auf ein relozierbares Symbol, das weder innerhalb des Segments CODE noch innerhalb des Segments DATA definiert ist.
- Ein Operand innerhalb der .OBJ-Datei bezieht sich auf eine als EXTRN deklarierte Prozedur oder Funktion über eine relative Adressangabe, wie beispielsweise der Befehl *CALL PascalSort+8*.

Kapitel 14 »Interne Details« des *Programmierhandbuches* enthält eine Auflistung der Regeln, die bei der Erstellung von Maschinenprogrammen zu beachten sind.

57 THEN expected
(**then** erwartet)

58 TO or DOWNTO expected
(**to** oder **downto** erwartet)

59 Undefined forward
(**forward**-Deklaration: Definition fehlt)
Diese Fehlermeldung kann zwei verschiedene Gründe haben:

- Die Prozedur oder Funktion wurde im Interface-Teil einer Unit deklariert, aber im Implementations-Teil nicht definiert.
- Die Prozedur oder Funktion wurde in einem Programm als **forward** deklariert, der Compiler hat aber die tatsächliche Definition nicht gefunden.

60 Too many procedures
(zuviele Prozeduren/Funktionen in einem Modul)
Turbo Pascal erlaubt maximal 512 Prozeduren und Funktionen in einem Programm oder einer Unit – Sie müssen eine zusätzliche Unit erstellen, in der Sie einige Routinen »auslagern« und diese Unit mit **uses** in das eigentliche Programm (oder die Unit) aufnehmen.

61 Invalid typecast

(Typ-Umwandlung nicht möglich)
Entweder stimmen die Größen von Original- und Ziel-Typ nicht überein oder Sie versuchen die Typ-Umwandlung eines Ausdrucks an einer Stelle, an der nur Variablen-Zugriffe stehen können.

62 Division by zero

(Division durch Null)
Der Compiler berechnet konstante Ausdrücke bereits während der Übersetzung und hat dabei offensichtlich eine Division durch Null vornehmen müssen.

63 Invalid file type

(Operation mit diesem Dateityp nicht erlaubt)
Die Operation kann auf eine Dateivariable des angegebenen Typs nicht angewendet werden (wie *Seek* auf eine Textdatei oder *Readln* auf eine typisierte Datei).

64 Cannot Read or Write Variables of this type

(Variablen dieses Typs können nicht gelesen/geschrieben werden)
Die Ausgabe von strukturierten Typen (außer Strings) und Zeigerwerten ist mit *Write* und *Writeln* nicht möglich, *Read* und *Readln* haben dieselben Grenzen. Boolsche Werte können mit *Write* und *Writeln* ausgegeben, nicht aber mit *Read* oder *Readln* eingelesen werden.

65 Pointer variable expected

(Zeiger-Variable erwartet)

66 String variable expected

(String-Variable erwartet)

67 String expression expected

(String-Ausdruck erwartet)

68 Circular unit reference

(überkreuzender Bezug zweier Units)
Sie haben zwei Units definiert, die sich gegenseitig im Interface-Teil voraussetzen, also beispielsweise:

```
unit U1;        unit U2;
uses U2;        uses U1;
```

Derartige Bezüge sind in Turbo Pascal nur über den Implementations-Teil erlaubt.

69 Unit name mismatch

(Unitname und Unit-Dateiname stimmen nicht überein)
Die Unit hat einen anderen Namen als die angegebene .TPU-Datei. (Der .TPU-Dateiname und der als **unit** deklarierte Bezeichner müssen grundsätzlich gleich sein).

70 Unit version mismatch
(Unit-Versionen stimmen nicht überein)
Eine oder mehrere Units, die von dieser Unit vorausgesetzt werden, sind seit der letzten Compilierung verändert worden. Verwenden Sie *Compile/Build* oder *Compile/Make* der integrierten Entwicklungsumgebung bzw. die Kommandozeilen-Parameter `/M` oder `/B`, damit Turbo Pascal diese Unit compiliert und damit auf den neusten Stand bringt.

71 Duplicate unit name
(Unit ist doppelt angegeben)
Die Unit ist in der **uses**-Anweisung mehrfach aufgeführt.

72 Unit file format error
(Linker: Unit-Dateiformat ungültig)
Der Linker kann den Inhalt der angegebenen Datei nicht als Unit interpretieren.

73 IMPLEMENTATION expected
(**implementation** erwartet)

74 Constant and case types do not match
(**case**-Konstante und Selektor haben verschiedene Datentypen)

75 Record variable expected
(Record-Variable erwartet)

76 Constant out of range
(Konstante außerhalb des zulässigen Wertebereichs)
Für diese Fehlermeldung kann es mehrere Gründe geben:

- Indizierung eines Arrays mit einer Konstanten, die außerhalb der Arraygrenzen liegt.
- Zuweisung eines (konstanten) Wertes an eine Variable, der außerhalb des Wertebereichs der Variablen liegt.
- Übergabe eines (konstanten) Parameters an eine Prozedur oder Funktion, dessen Wert außerhalb des Wertebereichs des entsprechenden formalen Parameters liegt.

77 File variable expected
(Datei-Variable erwartet)

78 Pointer expression expected
(Zeiger-Ausdruck erwartet)

79 Integer or real expression expected
(Integer- oder Real-Ausdruck erwartet)

80 Label not within current block
(**goto**-Sprungziel muß innerhalb des momentanen Blocks liegen)

81 Label already defined
(Label wurde bereits zur Markierung einer Anweisung verwendet)

82 Undefined label in preceding statement part
(vorangehender Anweisungsteil läßt Label undefiniert)
Innerhalb des vorangehenden Blocks wurde ein Label deklariert und auch als Sprungziel von **goto** verwendet; dem Compiler fehlt aber eine Definition, d.h. er kann nicht festlegen, wohin mit dem **goto** gesprungen werden soll.

83 Invalid @ argument
(unzulässiges Argument für **@**)
Als Argument für den Adress-Operator **@** können nur Variablenbezüge sowie Prozedur- und Funktionsbezeichner verwendet werden. **inline**-Prozeduren und Funktionen sind ebenfalls keine zulässigen Argumente für **@**.

84 UNIT expected
(**unit** erwartet)

85 ";" expected
(";" erwartet)

86 ":" expected
(":" erwartet)

87 "," expected
("," erwartet)

88 "(" expected
("(" erwartet)

89 ")" expected
(")" erwartet)

90 "=" expected
("=" erwartet)

91 ":=" expected
(":=" erwartet)

92 "[" or "(." expected
("[" oder "(." erwartet)

93 "]" or ".)" expected
("]" oder ".)" erwartet)

94 "." expected
("." erwartet)

95 ".." expected
(".." erwartet)

96 Too many variables
(zuviele Variablen)
Diese Fehlermeldung erhalten Sie, wenn entweder

- die Größe der globalen Variablen eines Programms und darin aufgenommener Units 64 KByte überschreitet, oder
- innerhalb einer Prozedur oder Funktion mehr als 64 KByte lokale Variablen deklariert sind.

Beide Fehlermöglichkeiten haben nichts miteinander zu tun: globale Variablen werden im Datensegment gespeichert, lokale Variablen im Stacksegment.

97 Invalid FOR control variable
(Variable nicht als Laufvariable einer **for**-Schleife verwendbar)
Als Laufvariable einer **for**-Schleife können nur einfache ordinale Variablen verwendet werden, die entweder global zum Programm oder lokal zum momentanen Block deklariert sein müssen.

98 Integer variable expected
(Integer-Variable erwartet)

99 File and procedure types are not allowed here
[Datei- und Prozduretypen sind hier nicht erlaubt.]

100 String length mismatch
(Stringlänge und Arraygröße stimmen nicht überein)
Die (absolute) Länge des Strings entspricht nicht der Anzahl der Elemente des angegebenen *Char*-Arrays.

101 Invalid ordering of fields
(Reihenfolge der Felder stimmt nicht überein)
Die Reihenfolge der Felder muß bei einer Record-Konstanten dieselbe sein wie bei der Definition des Recordtyps.

102 String constant expected
(String-Konstante erwartet)

103 Integer or real variable expected
(Integer- oder Real-Variable erwartet)

104 Ordinal variable expected
(Variable ordinalen Typs erwartet)

105 INLINE error
(**inline**: "<"-Operator nicht auf Variablen anwendbar)

106 Character expression expected
(Ausdruck des Ergebnistyps *Char* erwartet)

107 Too many relocation items
(Relozierungstabelle >64 KByte – Programm zu groß)
Die Relozierungstabelle der .EXE-Datei würde 64 KByte überschreiten (und damit die Obergrenze des Linkers von Turbo Pascal). Wenn Sie diese Fehlermeldung erhalten, ist Ihr Programm einfach zu groß für Turbo Pascal geworden – und wahrscheinlich auch zu groß für MS-DOS. In diesem Fall bleibt Ihnen nur die Unterteilung in ein »Hauptprogramm« und mehrere »Subprogramme« übrig, die jeweils über die Prozedur *Exec* (in der Unit *Dos*) gestartet werden.

112 CASE constant out of range
(**case**-Konstante außerhalb des zulässigen Bereichs)
Die Werte der **case**-Konstanten müssen im Bereich von -32768..32767 liegen, solange mit einem *Integer* als Selektor gearbeitet wird.

113 Error in statement
(Syntaxfehler in Befehl/Anweisung)
Mit dem von Ihnen angegebenen Symbol darf eine Anweisung nicht beginnen.

114 Cannot call an interrupt procedure
(direkter Aufruf von **interrupt**-Prozeduren ist nicht erlaubt)

116 Must be in 8087 mode to compile this
(Datentyp/Konstrukt nur im Modus `{$N+}` möglich)
Operationen mit den Variablentypen *Single*, *Double*, *Extended* und *Comp* setzen die Aufnahme der 8087-Bibliothek durch den Schalter `{$N+}` voraus. Wenn Ihr Computer nicht mit einem Coprozessor ausgerüstet ist, müssen Sie zusätzlich den Emulator mit dem Compiler-Befehl `{$E+}` einbinden.

117 Target address not found
(angegebene Adresse ist nicht Teil des Programms)
Die Suche nach einem Laufzeitfehler (mit *Compile/Find Error* bzw. dem Kommandozeilen-Parameter `/F`) konnte an der angegebenen Adresse keinen Befehl des Programms finden.

118 Include files are not allowed here
(Include-Befehl an dieser Stelle nicht erlaubt)
Der Compiler-Befehl `{$I <Dateiname>}` kann nicht inmitten eines Blocks gegeben werden.

120 NIL expected
(**nil** erwartet)

121 Invalid qualifier

(ungültige Qualifizierung)
Diese Fehlermeldung erhalten Sie bei dem Versuch,

- eine Variable zu indizieren, die kein Array ist;
- auf Felder einer Variablen zuzugreifen, die kein Record ist;
- über eine Variable *X*, die keinen Zeigertyp hat, auf eine andere Variable zuzugreifen (also mit *X*^).

122 Invalid variable reference

(ungültiger Variablenbezug)
Das angegebene Konstrukt folgt den Regeln eines Variablenbezugs, ergibt aber letztendlich nicht die Speicheradresse einer Variablen. Aller Wahrscheinlichkeit nach haben Sie eine Zeigerfunktion verwendet und vergessen, das Ergebnis dieser Funktion (mit ^) in eine Adresse umzuwandeln.

123 Too many symbols

(Symbole belegen mehr als 64 KByte Speicherplatz)
Das Programm oder die Unit deklariert soviele Symbole, daß 64 KByte Speicherplatz dafür nicht ausreichen. Abgesehen vom »Auslagern« einiger Prozeduren und Funktionen in eine separate Unit haben Sie die Möglichkeit, die Erzeugung von Symbolen mit dem Compiler-Befehl {*$D-*} zu unterdrücken – allerdings ist dann eine Zuordnung von Laufzeitfehlern oder eine schrittweise Ausführung mit dem integrierten Debugger nicht mehr möglich.

124 Statement part too large

(Anweisungsteil zu groß)
Turbo Pascal begrenzt den Anweisungsteil einzelner Prozeduren, Funktionen und des Hauptprogramms auf *jeweils* rund 24 KByte Code. Falls Sie diese Fehlermeldung erhalten, müssen Sie einige Befehlsfolgen in zusätzliche Prozeduren oder Funktionen verbannen. (Programme, bei denen ein durchgehender Anweisungsteil diese Grenze überschreitet, haben ein gründliches Aufräumen sowieso dringendst nötig).

126 Files must be var parameters

(Datei-Variablen müssen **var**-Parameter sein)
In den formalen Parameterlisten von Prozeduren und Funktionen müssen Variablen des Typs **File** grundsätzlich als **var**-Parameter deklariert sein, weil sie immer als Adresse übergeben werden.

127 Too many conditional symbols

(zuviele Symbole für die bedingte Compilierung)
Die Tabelle, in die der Compiler Symbole für die bedingte Compilierung einträgt, ist voll. Versuchen Sie, kürzere Symbolnamen zu verwenden und Symbole, die im weiteren Verlauf der Compilierung nicht mehr gebraucht werden, mit {*$UNDEF*} wieder zu löschen.

128 Misplaced conditional directive

({**$ELSE**} oder {**$ENDIF**} ohne vorangehendes {**$IF...**})

Der Compiler hat ein {$ELSE} oder {$ENDIF} gefunden, ohne daß zuvor eine Bedingung mit {$IFDEF}, {$IFNDEF} oder {$IFOPT} eingeleitet worden wäre.

129 ENDIF directive missing

({**$ENDIF**} fehlt)

Innerhalb der Quelltext-Datei wurde eine Bedingung mit {$IF...} begonnen, die am Ende der Datei nicht mit {$ENDIF} abgeschlossen worden ist. (Zu jedem {$IF...} gehört ein {$ENDIF} -Befehl innerhalb derselben Datei – eine bedingte Compilierung über die Grenzen von Include-Dateien hinweg ist nicht möglich).

130 Error in initial conditional defines

(Direkt definierte bedingte Symbole enthalten Fehler)

Die mit *O/C/Conditional defines* bzw. dem Kommandozeilen-Parameter /D vor dem Start der Compilierung definierten Symbole sind aus irgendeinem Grund ungültig. (Die zugrundeliegenden Regeln finden Sie in Kapitel 10 »Die Zusatzprogramme« des *Benutzerhandbuches*).

131 Header does not match previous definition

(Prozedur-/Funktions-Definition ungleich Deklaration)

Für diese Fehlermeldung gibt es zwei mögliche Gründe:

- Eine Funktion oder Prozedur führt im Interface-Teil einer Unit eine andere Parameterliste als im Implementations-Teil auf.
- Eine Funktion oder Prozedur führt bei der Deklaration als **forward** eine andere Parameterliste auf als bei ihrer Definition.

132 Critical disk error

(Diskettenfehler während der Compilierung)

Mögliche Gründe sind eine offene Laufwerksklappe, schadhafte Disketten usw.

133 Cannot evaluate this expression

(Dieser Ausdruck läßt sich in diesem Kontext nicht auswerten)

Diese Meldung kann zwei Gründe haben:

- Sie haben in einem Konstanten-Ausdruck eine Funktion der Laufzeitbibliothek verwendet, die der Compiler nicht für Konstanten zuläßt (z.B. *Sin*). Eine Beschreibung der Syntax und der verwendbaren Standardfunktionen finden Sie in Kapitel 5 »Typisierte Konstanten« des *Programmierhandbuches*.
- Sie haben versucht, eine Funktion des Programms in einem *Watch*-Ausdruck oder via *Break/Evaluate* aufzurufen.

134 Expression incorrectly terminated

(Ausdruck nicht korrekt abgeschlossen)

Der Compiler erwartet an dieser Stelle entweder den Abschluß eines Ausdrucks oder einen weiteren Operator – offensichtlich steht dort aber etwas anderes.

135 Invalid format specifier

(Format-Angabe nicht anwendbar/ungültig)

Diese Meldung erhalten Sie, wenn der Debugger mit der Format-Angabe in einem *Watch*-Ausdruck bzw. im Feld *Evaluate* von *Break/Evaluate* nichts anfangen kann.

136 Invalid indirect reference

(Unzulässiger indirekter Bezug)

Sie haben versucht, auf einen Bezeichner zuzugreifen, dessen Adresse dem Compiler unbekannt ist (wie z.B. bei einer Deklaration der Form **absolute** *xyz*, wobei das momentane Modul keinen Bezeichner *xyz* definiert). Ein weiterer Grund für diese Meldung sind Bezüge auf nicht definierte Variablen durch **inline**-Befehle.

137 Structured variables are not allowed here

(Strukturierte Variablen sind an dieser Stelle nicht erlaubt)

Mit dieser Meldung reagiert der Compiler auf Operatoren und Operationen, die sich nicht mit strukturierten Variablen durchführen lassen (wie z.B. den Versuch, zwei Records miteinander zu multiplizieren oder ein Array mit *Writeln* auszugeben).

138 Cannot evaluate without system unit

(Ohne SYSTEM läuft hier gar nichts – nicht einmal der Debugger)

Mit dieser Meldung reagiert der Debugger auf den Versuch, einen Ausdruck zu berechnen, wenn Sie die Unit SYSTEM aus TURBO.TPL entfernt haben.

139 Cannot access this symbol

(Bezeichner/Wert ist nicht verfügbar)

Um den Wert einer Variablen mit dem Debugger zu ermitteln, muß zumindest der erste Schritt des Hauptprogramms ausgeführt werden. (Dieser Schritt enthält die Initialisierungsteile aufgenommener Units, über die unter anderem das Datensegment-Register gesetzt wird).

140 Invalid floating point operation

(Gleitkomma-Rechenfehler)

Mit dieser Meldung reagiert der Compiler, wenn die Bewertung einer Konstanten mit einer Division durch Null oder einem Überlauf endet.

141 Cannot compile overlays to memory

(Overlay-Units können nur als Diskettendateien compiliert werden)

Mit dieser Meldung reagiert der Compiler auf jeden Befehl der Form {*$O Unitname*}, wenn *Compile/Destination* nicht auf *Disk* gesetzt ist, weil die Overlay-Verwaltung von Turbo Pascal grundsätzlich eine .OVR-Datei auf der Diskette erwartet. (Units, die später als Overlays *verwendet* werden sollen, lassen sich allerdings – beispielsweise zur Syntaxprüfung – jederzeit mit *Destination.. Memory* übersetzen).

142 Procedure or function variable expected

(Prozedur- oder Funktions-Variable erwartet)

Der Compiler erwartet an dieser Stelle den Bezeichner einer Prozedur bzw. Funktion (beispielsweise, wenn auf der linken Seite einer Zuweisung eine Prozedur-Variable steht).

143 Invalid procedure or function reference

(Bezug auf Prozedur/Funktion in dieser Weise nicht zulässig)

Diese Meldung kann zwei recht verschiedene Gründe haben:

- der Name einer Prozedur erscheint innerhalb eines Ausdrucks;
- der Compiler moniert die Zuweisung einer Routine an eine Prozedur-Variable: die Routine wurde entweder nicht mit `{$F+}` übersetzt, ist als **inline** oder **interrupt** deklariert oder stammt aus der Laufzeitbibliothek (vgl. Kapitel 4 »Variablen« des *Programmierhandbuches*).

144 Cannot overlay this unit

(Diese Unit läßt sich nicht als Overlay deklarieren)

Sie haben versucht, eine Unit mit `{$O Unitname}` als Overlay zu deklarieren, die nicht im Modus `{$O+}` compiliert wurde,.

145 Too many nested scopes

(Zuviele Verschachtelungsebenen)

Jede mit **uses** aufgeführte Unit, jede Record-Deklaration und jede **with**-Anweisung werden vom Compiler als eingeschachtelte Ebene gewertet. Eine Anweisung wie

```
with RecX do
begin
  with Field.X do ....   { = RecX.FieldX }
```

enthält also zwei Ebenen; befindet sie sich in einer Unit, die mit **uses** in ein Programm aufgenommen wurde, sind es insgesamt drei Ebenen; wurde die Unit dagegen von einer anderen Unit mit **uses** aufgenommen, sind es vier usw.

Turbo Pascal wird mit 128 Ebenen pro Unit und maximal 512 Ebenen pro Projekt fertig – es ist also mehr als unwahrscheinlich, daß Sie diese Fehlermeldung jemals zu Gesicht bekommen.

146 File access denied

[Zugriff auf Datei verweigert.]

147 Object type expected.

(Objekttyp erwartet.)

Die Variablenbezeichnung steht nicht für einen Objekttyp.

148 Local object types are not allowed.

(Lokale Objekttypen sind nicht zulässig.)

Objekttypen dürfen nur im äußersten Bezugsrahmen eines Programms oder

einer Unit definiert werden. Die Definition eines Objekttyps innerhalb von Prozeduren und Funktionen ist nicht zulässig.

149 VIRTUAL expected.
(VIRTUAL erwartet.)
Das reservierte Wort **virtual** fehlt.

150 Method identifier expected.
(Bezeichner für Methode erwartet.)
Der angegebene Bezeichner ist keine Methode.

151 Virtual constructors are not allowed.
(Virtuelle Konstruktoren sind nicht zugelassen.)
Die Methode für einen Konstruktor muß statisch sein.

152 Constructor identifier expected.
(Bezeichner für einen Konstruktor erwartet.)
Der angegebene Bezeichner ist kein Konstruktor.

153 Destructor identifier expected.
(Bezeichner für einen Destruktor erwartet.)
Der angegebene Bezeichner ist kein Destruktor.

154 Fail only allowed within constructors.
(Fail darf nur innerhalb von Kontruktoren verwendet werden.)
Die Standardprozedur **Fail** darf nur innerhalb eines Konstruktors verwendet werden.

155 Invalid combination of opcode and operands.
(Ungültige Kombination von Opcodes und Operanden)
Der Assembler-Opcode akzepitert diese Operandenkombination nicht. Mögliche Ursachen dafür sind:

- Es wurden zu viele bzw. zu wenige Operanden für einen Assembler-Opcode angegeben, z.B. **INC AX,BX** oder **MOV AX**
- Die Anzahl der Operanden ist zwar korrekt, ihr Typ und/oder ihre Reihenfolge ist jedoch falsch, z.B. **DEC 1**, **MOV AX,CL** oder **MOV 1,AX**

156 Memory reference expected.
(Speicherreferenz erwartet)
Der Assembler-Operrand stellt keine hier benötigte Speicherreferenz dar. In der Regel haben Sie lediglich vergessen, eckige Klammern um einen Index-Register-Operanden zu setzen, z.B. **MOV AX,BX+SI** anstelle von **MOV AX,[BX+SI]**.

157 Cannot add or substract relocatable symbols.
(Kann verschiebbares Symbol nicht addieren oder subtrahieren)
Die einzige arithmetische Operation, die an einem verschiebbaren Symbol in

einem Assembler-Operanden vorgenommen werden kann, ist die Addition oder Subtraktion von Konstanten. Variablen, Prozeduren, Funktionen und Labels sind verschiebbare Symbole.

Ein Beispiel: *Var* sei eine Variable und *Const* eine Konstante, dann sind **MOV AX, Const+Const** sowie **MOV AX, Var+Const** gültige Ausdrücke, der Ausdruck **MOV AX, Var+Var** hingegen ist ungültig.

158 Invalid register combination.

(Ungültige Kombination von Registern)

Gültige Kombinationen von Index-Registern sind: **[BX]**, **[BP]**, **[SI]**, **[DI]**, **[BX+SI]**, **[BX+DI]**, **[BP+SI]** und **[BP+DI]**. Andere Kombinationen von Index-Registern, wie **[BP+BX]** und **[SI+DI]** sind nicht erlaubt.

Hinweis: Lokale Variablen (Variablen, die in Prozeduren bzw. Funktionen deklariert sind) werden immer dem Stack zugeordnet, der Zugriff auf sie erfolgt über das BP-Register.

159 286/287 Instructions are not enabled.

(286/287 Anweisungen sind nicht zugelassen)

Aktivieren Sie mit dem Compiler-Schalter {*$G+*} die Erzeugung der 286/287-Opcodes. Bedenken Sie dabei jedoch, daß ein auf diese Weise compiliertes Programm auf Rechnern mit 8086- oder 8088-Prozessoren nicht mehr lauffähig ist.

160 Invalid symbol reference.

(Ungültige Symbol-Referenz)

Auf dieses Symbol kann in einem Assembler-Operanden nicht zugegriffen werden. Mögliche Gründe dafür sind:

- Sie versuchen in einem Assembler-Operanden auf eine Standard-Prozedur bzw. -Funktion oder auf die speziellen Arrays *Mem*, *MemW*, *MemL*, *Port* oder *PortW* zuzugreifen.
- Sie versuchen in einem Assembler-Operanden auf eine String-, Gleitkomma- oder Mengen-Konstante zuzugreifen.
- Sie versuchen in einem Assembler-Operanden auf eine **Inline**-Prozedur oder -Funktion zuzugreifen.
- Sie versuchen außerhalb einer Funktion auf das *@Result*-Symbol zuzugreifen.
- Sie versuchen eine short **JMP**-Anweisung zu erzeugen, die nicht auf ein Label verzweigt.

161 Code generation error.

(Fehler bei Code-Erzeugeung)

Der vorangehende Anweisungsteil enthält eine LOOPNE-, LOOPE-, LOOP- oder JCXZ-Anweisung, deren Ziel-Label nicht erreichbar ist.

162 ASM expected.

(ASM erwarted)

Laufzeitfehler

Turbo Pascal unterscheidet bei Fehlern, die während der Ausführung eines Programms auftreten, zwei Klassen:
Ein-/Ausgabefehler (I/O Errors) haben Codes im Bereich von 1..199 und können über den Compiler-Befehl `{$I-}` abgefangen werden (siehe nächster Abschnitt).
Fehler mit sofortigem Abbruch (Fatal Errors) haben Codes im Bereich von 200..255 und einen unbedingten Abbruch zur Folge; programmgesteuerte Korrekturmaßnahmen sind hier nicht möglich.

Beim Abbruch durch einen Laufzeitfehler erhalten Sie (unabhängig von der Fehlerklasse) eine Meldung des Formats

```
Runtime error nnn at xxxx:yyyy
```

nnn steht dabei für den Fehlercode, *xxxx:yyyy* für die Adresse des Befehls, der den Fehler ausgelöst hat (*xxxx* ist die Segmentadresse, *yyyy* der Offset-Anteil).

Ein-/Ausgabefehler

Normalerweise codiert der Compiler nach jedem Ein-/Ausgabebefehl den Aufruf einer Prüfroutine, die im Falle eines Fehlers das Programm abbricht. Ein Programmteil, dem der Compiler-Befehl `{$I-}` vorangestellt ist, wird dagegen auch im Fehlerfall fortgesetzt – hier muß der Programmierer durch einen Aufruf der Funktion *IOResult* explizit prüfen, ob die Operation fehlerfrei verlaufen ist.

Das Ergebnis 0 steht für »fehlerfreie Ausführung«; andere mögliche Ergebnisse von *IOResult* lassen sich in vier Klassen einteilen:

1.. 99	Fehlercodes von DOS
100..149	Fehler bei der Bearbeitung von Dateien
150..199	»Critical Errors« (Laufwerksklappe offen, kein Papier im Drucker usw.)
200..249	Fehler mit sofortigem Abbruch des Programms

Fehlercodes von DOS

2 File not found
(Datei nicht gefunden)
Ausgelöst von *Reset*, *Append*, *Rename* und *Erase*, wenn der der Datei-Variablen zugeordnete Name keine existierende Datei bezeichnet bzw. diese Datei nicht innerhalb des momentanen/angegebenen Verzeichnisses zu finden ist.

3 Path not found
(Suchweg nicht gefunden)

- Ausgelöst von *Reset*, *Rewrite*, *Append*, *Rename* und *Erase*, wenn der der Datei-Variablen zugeordnete Name einen ungültigen oder nicht existierenden Suchweg angibt.

- Ausgelöst von *ChDir*, *MkDir* und *RmDir*, wenn der angegebene Suchweg ungültig ist oder ein nicht existierendes Verzeichnis bezeichnet.

4 Too many open files

(Maximalzahl an Dateien bereits offen)

Ausgelöst von *Reset*, *Rewrite* und *Append*, wenn das System die Eröffnung einer weiteren Datei aufgrund fehlender Dateipuffer verweigert. Maximal sind 15 Diskettendateien pro Prozeß erlaubt. Falls Sie diesen Fehler früher erhalten, sollten Sie die Datei CONFIG.SYS prüfen und gegebenenfalls um den Befehl `FILES=20` erweitern.

Hinweis: Damit Veränderungen von CONFIG.SYS wirksam werden, müssen Sie den Computer neu starten.

5 File access denied

(Dateizugriff verweigert)

- Ausgelöst von *Reset* und *Append*, wenn das Feld *FileMode* der Datei-Variablen Schreib- und Lesezugriffe erlaubt ist und die Datei-Variable einer gegen Überschreiben gesperrten Datei oder einem (Unter-)Verzeichnis zugeordnet ist.

- Ausgelöst von *Rewrite*, wenn die Datei-Variable einem (Unter-)Verzeichnis zugeordnet ist oder das entsprechende Verzeichnis bei der Erzeugung einer neuen Datei keinen Platz für den Dateieintrag bietet.

- Ausgelöst von *Rename*, wenn der der Datei-Variablen zugeordnete Name ein (Unter-)Verzeichnis bezeichnet oder bereits eine Datei mit dem neuen Namen existiert.

- Ausgelöst von *Erase*, wenn der der Datei-Variablen zugeordnete Name ein (Unter-)Verzeichnis oder eine gegen Schreibzugriffe gesperrte Datei bezeichnet.

- Ausgelöst von *MkDir*, wenn bereits eine »normale« Datei des angegebenen Namens innerhalb des Verzeichnisses existiert, in dem das SubVerzeichnis eingetragen werden soll, das Verzeichnis keinen Platz für einen weiteren Eintrag bietet oder der angegebene Suchweg ein Peripheriegerät bezeichnet.

- Ausgelöst von *RmDir*, wenn das angegebene Verzeichnis nicht leer ist (d.h. noch Dateieinträge enthält), oder wenn es sich beim angegebenen Suchweg um eine »normale« Datei oder um das Stammverzeichnis des Datenträgers handelt.

- Ausgelöst von *Read* oder *BlockRead* bei einer typisierten bzw. untypisierten Datei, wenn diese Datei nicht für Leseoperationen geöffnet wurde.

- Ausgelöst von *Write* oder *BlockWrite* bei einer typisierten bzw. untypisierten Datei, wenn diese Datei nicht für Schreiboperationen geöffnet wurde.

6 Invalid file handle

(Handle nicht definiert/ungültig)

Dieser Fehler sollte theoretisch nicht vorkommen (solange mit Pascal-Prozeduren gearbeitet wird) – falls doch, ist die Datei-Variable aus irgendeinem Grund überschrieben bzw. zerstört worden.

12 Invalid file access mode
(ungültiger Dateimodus)
Ausgelöst von *Reset* und *Append* bei der Anwendung auf typisierte und untypisierte Dateien, wenn der angegebene Zugriff nicht möglich ist, weil das Feld *FileMode* einen ungültigen Wert hat.

15 Invalid drive number
(Laufwerksnummer unzulässig)
Ausgelöst von *GetDir*, wenn die angegebene Laufwerksnummer außerhalb der zulässigen Grenzen liegt.

16 Cannot remove current directory
(als Standard gesetztes Directory kann nicht gelöscht werden)
Ausgelöst von *RmDir* beim Versuch, das aktuell gesetzte Verzeichnis zu löschen.

17 Cannot rename across drives
(Rename kann nicht kopieren)
Ausgelöst von *Rename*, wenn für den alten und den neuen Dateinamen zwei verschiedene Laufwerke angegeben werden.

Fehler bei der Bearbeitung von Dateien

100 Disk read error
(Fehler beim Lesen von der Diskette)
Ausgelöst von *Read* beim Versuch, über das Ende einer typisierten Datei hinaus weitere Daten zu lesen.

101 Disk write error
(Fehler beim Schreiben auf die Diskette)
Ausgelöst von *Close*, *Write*, *Writeln*, *Flush* und *Page*, wenn die Diskette voll ist.

102 File not assigned
(Datei-Variable ist keiner Datei zugeordnet)
Ausgelöst von *Reset*, *Rewrite*, *Append*, *Rename* und *Erase*, wenn die Datei-Variable noch nicht (durch einen Aufruf von *Assign*) einer Datei zugeordnet wurde.

103 File not open
(Datei ist nicht offen)
Ausgelöst von *Close*, *Read*, *Write*, *Seek*, *Eof*, *FilePos*, *FileSize*, *Flush*, *BlockRead* und *BlockWrite* bei der Anwendung auf eine Datei, die noch nicht eröffnet worden ist.

104 File not open for input
(Datei wurde nicht für Leseoperationen geöffnet)
Ausgelöst von *Read*, *Readln*, *Eof*, *Eoln*, *SeekEof* und *SeekEoln* bei der Anwendung auf eine Textdatei, die nur für Schreiboperationen geöffnet wurde.

105 File not open for output
(Datei wurde nicht für Schreiboperationen geöffnet)
Ausgelöst von *Write*, *Writeln* und *Page* bei der Anwendung auf eine Textdatei, die nur für Leseoperationen geöffnet wurde.

106 Invalid numeric format
(ungültiges numerisches Format)
Ausgelöst von *Read* oder *Readln*, wenn von einer Textdatei gelesene Werte nicht dem numerischen Format der angegebenen Variablen entsprechen.

Critical Errors

150 Disk is write protected
(Diskette ist schreibgeschützt)

151 Unknown Unit
(Peripheriegerät nicht bekannt/nicht angeschlossen)

152 Drive not ready
(Laufwerk nicht betriebsbereit)

153 Unknown command
(ungültiger DOS-Funktionscode/Funktion nicht definiert)

154 CRC error in data
(Prüfsummen-Fehler beim Lesen von der Diskette/Fesplatte)

155 Bad drive request structure length
(ungültiger Disk-Parameterblock)

156 Disk seek error
(Kopf-Positionierungsfehler auf der Diskette/Festplatte)

157 Unknown media type
(unbekanntes Sektorformat)

158 Sector not found
(Diskettensektor nicht lokalisierbar)

159 Printer out of paper
(kein Papier im Drucker)

160 Device write fault
(Schreibfehler beim Zugriff auf ein Peripheriegerät)

161 Device read fault
(Lesefehler beim Zugriff auf ein Peripheriegerät)

162 Hardware failure
(nicht genauer bestimmbarer Hardware-Fehler)

Fehler mit sofortigem Abbruch (Fatal Errors)

200 Division by zero
(Division durch Null)

201 Range check error
(Bereichsüberprüfung: Fehler entdeckt)
Diesen Fehler erhalten Sie, wenn das Programm im Modus {*$R+*} compiliert wurde und eine der folgenden Bedingungen zutrifft:

- der Index-Ausdruck bei einem Zugriff auf ein Array gibt ein nicht existierendes Element an (d.h. sein Wert überschreitet die Array-Grenzen);
- das Programm hat versucht, einer Variablen einen Wert außerhalb ihres zulässigen Bereichs zuzuweisen;
- das Programm hat versucht, einer Prozedur oder Funktion einen Wert zu übergeben, der außerhalb des zulässigen Wertebereichs des entsprechenden formalen Parameters dieser Routine liegt.

202 Stack overflow error
(Stacküberprüfung: Überlauf entdeckt)
Diesen Fehler erhalten Sie, wenn das Programm im Modus {*$S+*} compiliert wurde und es sich nun beim Aufruf einer Prozedur oder Funktion herausstellt, daß nicht mehr genügend Platz auf dem Stack vorhanden ist. (Solange es sich nicht um einen Programmierfehler handelt: verwenden Sie den Compiler-Befehl {*$M*}, um die Größe des Stacks entsprechend zu erhöhen.)

203 Heap overflow error
(kein Platz mehr im Heap-Bereich)
Dieser Fehler wird von *New* und *GetMem* ausgelöst, wenn nicht mehr genug Platz auf dem Heap ist, um einen Speicherbereich der gewünschten Größe zu belegen. (Details inklusive der Möglichkeit, diese Art von Fehlern abzufangen, finden Sie in Kapitel 14 »Interne Details« des *Programmierhandbuches*).

204 Invalid pointer operation
(ungültige Zeiger-Operation)
Dieser Fehler wird von *Dispose* oder *FreeMem* erzeugt, wenn die übergebene

Zeigervariable den Wert **nil** hat, auf eine Adresse außerhalb des Heap zeigt oder die Fragmentliste keine weiteren Einträge mehr aufnehmen kann (siehe Kapitel 14 »Interne Details« des *Programmierhandbuches*).

205 Floating point overflow
(Gleitkomma-Überlauf)

206 Floating point underflow
(Gleitkomma-Unterlauf)
Unterläufe liefern normalerweise den Wert 0 – diesen Fehler erhalten Sie nur bei Verwendung eines numerischen Coprozessors und der Freigabe des Bits für Unterläufe im Statusregister des 8087.

207 Invalid floating point operation
(Gleitkomma-Fehler)

- Ausgelöst von *Trunc* und *Round*, wenn der übergebene Real-Wert nicht innerhalb der Bereichsgrenzen von *LongInt* (-2147483648.. 2147483647) in einen ganzzahligen Wert konvertiert werden konnte.
- Ausgelöst von *Sqrt* bei Aufrufen mit negativem Argument.
- Ausgelöst von *Ln* bei Aufrufen mit negativem Argument oder dem Wert 0.
- Ausgelöst durch einen Stack-Überlauf des 8087 (mehr als acht Zwischenergebnisse bzw. Überlauf durch Rekursion – siehe Kapitel 13 »Der Coprozessor 8087« des *Programmierhandbuches*).

208 Overlay manager not installed
(Overlay-Verwaltung nicht installiert)
Das Programm hat versucht, vor der Initialisierung der Overlay-Verwaltung eine Overlay-Routine aufzurufen. Entweder fehlt der für Overlay-Programme obligatorische Aufruf von *OvrInit*, oder diese Funktion konnte aus irgendeinem Grund nicht fehlerfrei beendet werden (Overlay-Datei nicht gefunden, unzureichender Speicherplatz etc.).

Eine weitere Möglichkeit besteht darin, daß eine der Overlay-Units einen Initialisierungsteil definiert – in diesem Fall muß eine statische Unit programmiert werden, die *vor* der entsprechenden Overlay-Unit mit **uses** angegeben wird und deren Initialisierung die Overlay-Verwaltung startet (siehe Kapitel 3 »Units,Overlays und andere Geheimnisse« des *Benutzerhandbuches*).

209 Overlay file read error
(I/O-Fehler beim Lesen der Overlay-Datei)
Mit diesem Code wird ein Programm abgebrochen, wenn die Overlay-Verwaltung ein Overlay nicht korrekt von der Diskette lesen kann. (Der Abbruch erfolgt nur dann, wenn die entsprechende Unit tatsächlich benötigt wird – *OvrInit* und *OvrInitEMS* setzen im Fehlerfall lediglich die Statusvariable *OvrError*).

Der folgende Laufzeitfehler erscheint, wenn die Bereichsüberprüfung mit `{R+}` oder durch die entsprechende Compiler-Option eingeschaltet wurde:

210 Object not initialized

[Objekt nicht initialisiert.]

Sie haben versucht, die virtuelle Methode eines Objektes aufzurufen, obwohl dieses Objekt noch nicht (mittels eines Konstruktor-Aufrufs) initialisiert ist.

211 Call to abstract method

(Aufruf einer abstrakten Methode)

Dieser Fehler wird von der Prozedur *Abstract* in der Unit *Objects* ausgelöst und zeigt an, daß Ihr Programm versucht hat, eine abstrakte virtuelle Methode aufzurufen. Enthält ein Objekt eine oder mehrere abstrakte Methoden, dann handelt es sich bei diesem Objekt um einen *abstrakten Objekttyp*, der nicht instantiiert werden darf. Sie müssen ein neues Objekt von diesem abstrakten Objekttyp ableiten und dessen abstrakte Methoden überschreiben. Zum Beispiel ist die Methode *Compare* des Objekts *TSortedCollection* in der Unit *Objects* abstrakt. Sie müssen also von *TSortedCollection* ein neues Objekt ableiten und die Methode *Compare* überschreiben.

212 Stream registration error

(Fehler bei Registrierung eines Objekts)

Dieser Fehler wird von der Prozedur *RegisterType* in der Unit *Objects* ausgelöst und zeigt an, daß einer der folgenden Fehler vorliegt:

- Der Stream-Registrationsrecord befindet sich nicht im Datensegment.
- Das Feld *ObjType* des Stream-Registrationsrecords enthält den Wert Null.
- Der Typ wurde schon registriert.
- Es existiert schon ein anderer Typ mit dem gleichen Wert in *ObjType*.

213 Collection index out of range

(Index außerhalb des zulässigen Bereichs)

Der Index, der beim Aufruf einer Methode von *TCollection* übergeben wurde, liegt außerhalb des zulässigen Bereichs.

214 Collection overflow error

(Überlauf einer Kollektion)

Dieser Fehler wird von *TCollection* ausgelöst, wenn Sie versuchen, einer Kollektion, die nicht mehr erweitert werden kann, ein Element hinzuzufügen.

4. Zeichensatz und Tastaturcodes

ASCII-Codes

Intern stellt der Computer Zeichen (wie allen anderen Daten auch) durch Zahlenwerte dar. Der Zusammenhang zwischen Zeichen und dazugehörigen Nummern wird durch den *American Standard Code for Information Interchange* (= amerikanischer Standard-Code zum Informationsaustausch, ASCII) festgelegt. Ursprünglich war dieser Code nur mit 7 Bit (=128 Zeichen) definiert - die Firma IBM hat sich die Freiheit genommen, ihn um ein zusätzliches Bit zu erweitern (d.h. insgesamt auf 256 Zeichen).

Die Tabelle 4.1 auf den nächsten beiden Seiten gibt die Zeichen des ASCII-Codes sowie seiner Erweiterung durch IBM wieder.

Tabelle 4.1: Der ASCII-Zeichensatz und Erweiterungen

DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR
0	0		43	2B	+	86	56	V
1	1	☺	44	2C	,	87	57	W
2	2	☻	45	2D	-	88	58	X
3	3	♥	46	2E	.	89	59	Y
4	4	♦	47	2F	/	90	5A	Z
5	5	♣	48	30	0	91	5B	[
6	6	♠	49	31	1	92	5C	\
7	7	•	50	32	2	93	5D	]
8	8	◘	51	33	3	94	5E	^
9	9	○	52	34	4	95	5F	_
10	A	◙	53	35	5	96	60	‘
11	B	♂	54	36	6	97	61	a
12	C	♀	55	37	7	98	62	b
13	D	♪	56	38	8	99	63	c
14	E	♫	57	39	9	100	64	d
15	F	¤	58	3A	:	101	65	e
16	10	▶	59	3B	;	102	66	f
17	11	◀	60	3C	<	103	67	g
18	12	↕	61	3D	=	104	68	h
19	13	‼	62	3E	>	105	69	i
20	14	¶	63	3F	?	106	6A	j
21	15	§	64	40	@	107	6B	k
22	16	■	65	41	A	108	6C	l
23	17	↨	66	42	B	109	6D	m
24	18	↑	67	43	C	110	6E	n
25	19	↓	68	44	D	111	6F	o
26	1A	→	69	45	E	112	70	p
27	1B	←	70	46	F	113	71	q
28	1C	∟	71	47	G	114	72	r
29	1D	↔	72	48	H	115	73	s
30	1E	▲	73	49	I	116	74	t
31	1F	▼	74	4A	J	117	75	u
32	20		75	4B	K	118	76	v
33	21	!	76	4C	L	119	77	w
34	22	"	77	4D	M	120	78	x
35	23	#	78	4E	N	121	79	y
36	24	$	79	4F	O	122	7A	z
37	25	%	80	50	P	123	7B	{
38	26	&	81	51	Q	124	7C	|
39	27	’	82	52	R	125	7D	}
40	28	(	83	53	S	126	7E	~
41	29	)	84	54	T	127	7F	Δ
42	2A	*	85	55	U	128	80	Ç

Tabelle 4.1: Der ASCII-Zeichensatz und Erweiterungen (Fortsetzung)

DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR
129	81	ü	172	AC	¼	215	D7	╫
130	82	é	173	AD	¡	216	D8	╪
131	83	â	174	AE	«	217	D9	┘
132	84	ä	175	AF	»	218	DA	┌
133	85	à	176	B0	▒	219	DB	▉
134	86	å	177	B1	▒	220	DC	▄
135	87	ç	178	B2	▓	221	DD	▌
136	88	ê	179	B3	│	222	DE	▐
137	89	ë	180	B4	┤	223	DF	▀
138	8A	è	181	B5	╡	224	E0	α
139	8B	ï	182	B6	╢	225	E1	β
140	8C	î	183	B7	╖	226	E2	Γ
141	8D	ì	184	B8	╕	227	E3	π
142	8E	Ä	185	B9	╣	228	E4	Σ
143	8F	Å	186	BA	║	229	E5	σ
144	90	É	187	BB	╗	230	E6	µ
145	91	æ	188	BC	╝	231	E7	τ
146	92	Æ	189	BD	╜	232	E8	ϕ
147	93	ô	190	BE	╛	233	E9	θ
148	94	ö	191	BF	┐	234	EA	Ω
149	95	ò	192	C0	└	235	EB	δ
150	96	û	193	C1	┴	236	EC	∞
151	97	ù	194	C2	┬	237	ED	∅
152	98	ÿ	195	C3	├	238	EE	ε
153	99	Ö	196	C4	─	239	EF	∩
154	9A	Ü	197	C5	┼	240	F0	≡
155	9B	¢	198	C6	╞	241	F1	±
156	9C	£	199	C7	╟	242	F2	≥
157	9D	¥	200	C8	╚	243	F3	≤
158	9E	Pt	201	C9	╔	244	F4	⌠
159	9F	*ƒ*	202	CA	╩	245	F5	⌡
160	A0	á	203	CB	╦	246	F6	÷
161	A1	í	204	CC	╠	247	F7	≈
162	A2	ó	205	CD	═	248	F8	°
163	A3	ú	206	CE	╬	249	F9	∙
164	A4	ñ	207	CF	╧	250	FA	·
165	A5	Ñ	208	D0	╨	251	FB	√
166	A6	ª	209	D1	╤	252	FC	^{n}
167	A7	º	210	D2	╥	253	FD	^{2}
168	A8	¿	211	D3	╙	254	FE	■
169	A9	⌐	212	D4	╘	255	FF	
170	AA	¬	213	D5	╒			
171	AB	½	214	D6	╓			

Erweiterte Tastaturcodes

Über die Tastatur des Rechners lassen sich zwei Arten von Zeichen erzeugen:

- Den auf der vorhergehenden Seite aufgelisteten ASCII-Zeichensatz, der (in seiner von 0-127 auf 0-255 erweiterten Form) jedes Zeichen durch ein Byte repräsentiert. Das Lesen eines solchen Zeichens via *ReadKey* liefert folgerichtig auch den ASCII-Codes des eingegebenen Zeichens zurück.

- Die sogenannten erweiterten Tastaturcodes, die durch Spezialtasten oder bestimmte Tastenkombinationen eingegeben werden. Der erste Aufruf von *ReadKey* nach der Eingabe eines derartigen Zeichens liefert ein NUL-Zeichen (ASCII-Code 0) zurück, der zweite Aufrufe in den meisten Fällen den sogenannten *Scancode* wieder. Tabelle 4.2 listet den Zusammenhang zwischen und eingegebenen Tasten(-kombinationen) und dem sich ergebenden zweiten Byte auf, wobei national unterschiedliche Zeichen (wie z.B. Alt [) bewußt ausgelassen sind.

Tabelle 4.2: Erweiterte Tastaturcodes

2. Byte	Taste/Kombination
3	NUL-Zeichen
15	Shift ↓
16-25	Alt Q W E R T Y U I O P
30-38	Alt A S D F G H J K L
44-50	Alt Z X C V B N M
59-68	F1 .. F10
71	Home
72	←
73	PgUp
75	↑
77	↓
79	End
80	→
81	PgDn
82	Ins
83	Del
84-93	F11 .. F20 (Shift F1 bis Shift F10)
94-103	F21 F30 (Ctrl F1 bis Ctrl F10)
104-113	F31 .. F40 (Alt F1 bis Alt F10)
114	Ctrl PrtSc
115	Ctrl ↑
116	Ctrl ↓
117	Ctrl End
118	Ctrl PgDn
119	Ctrl Home

2. Byte	Taste/Kombination
120-129	Alt 1 2 3 4 5 6 7 8 9 0
131	Ctrl PgUp
133	F11
134	F12
135	Shift F11
136	Shift F12
137	Ctrl F11
138	Ctrl F12
139	Alt F11
140	Alt F12

Scancodes der Tastatur

Die Umwandlung einer Tastenkombination wie Alt F4 in einen erweiterten Tastaturcode wie `chr(0) + chr(107)` findet natürlich nicht durch die Tastatur selbst, sondern durch eine entsprechende BIOS-Routine statt – die Tastatur liefert lediglich die entsprechenden Tastendrücke in Form eines Scancodes zurück (in unserem Beispiel einen Scancode für die Taste Alt und einen zweiten Code für die Taste F4). Bei einem direkten Zugriff auf die Tastatur erzeugen also auch Modifizierer wie Shift , Alt und Ctrl eigene Codes. (Nur deshalb kann ein Programm wie SideKick auf Tastenkombinationen wie Shift Shift reagieren). Die folgende Tabelle gibt die Scancodes der einzelnen Tasten in hexadezimaler Notation wieder.

Tabelle 4.3 : Scancodes der Tastatur

Taste	Scan Code in Hex	Taste	Scan Code in Hex
Esc	01	Tabulator	0F
!1	02	Q	10
"2	03	W	11
§3	04	E	12
$4	05	R	13
%5	06	T	14
&6	07	Z	15
/7	08	U	16
(8	09	I	17
)9	0A	O	18
=0	0B	P	19
?ß	0C	Ü	1A
‘’	0D	*+	1B

Taste	Scan Code in Hex	Taste	Scan Code in Hex
Backspace	0E	Return	1C
Ctrl	1D	^#	2B
A	1E	Y	2C
S	1F	X	2D
D	20	C	2E
F	21	V	2F
G	22	B	30
H	23	N	31
J	24	M	32
K	25	;,	33
L	26	:.	34
Ö	27	_-	35
Ä	28	Shift-rechts	36
><	29	Prtsr*	37
Shift-links	2A	Alt	38
Leertaste	39	7 Home	47
Caps Lock	3A	8 Aufwärtspfeil	48
F1	3B	9 PgUp	49
F2	3C	Minuszeichen	4A
F3	3D	4 Linkspfeil	4B
F4	3E	5	4C
F5	3F	6 Rechtspfeil	4D
F6	40	+	4E
F7	41	1 End	4F
F8	42	2 Abwärtspfeil	50
F9	43	3 PgDn	51
F10	44	0 Ins	52
F11	D9	Del	53
F12	DA	NumLock	45
ScrollLock	46		

Index

A
{$A+/-} 180
Abs 3
Abstrakte Methode aufgerufen 219
Addr 4
Aktuelles Verzeichnis nicht löschbar 215
Append 4
Arc 5
ArcTan 6
Assign 6
AssignCrt 7
Ausrichtung von Variablen 180
Auswertung boolscher Ausdrücke 181

B
{$B+/-} 181
Bar 8
Bar3D 9
Bedingte Compiler-Direktive falsch 208
Bereichs-Überprüfung 185
Bereichsüberprüfung-Fehler 217
Bezeichner doppelt 195
Bezeichner erwartet 195
Bezeichner unbekannt 195
BlockRead 10
BlockWrite 11

C
CASE-Konstante außerhalb Bereich 206
ChDir 12
Chr 13
Circle 13
ClearDevice 14
ClearViewPort 14
Close 15
CloseGraph 16
ClrEol 16
ClrScr 17
Code-Erzeugungsfehler 212
Codesegment zu groß 200
Compiler-Fehlermeldungen 193
Concat 17
Copy 18
Cos 18
CRC-Fehler bei Daten 216
Critical disk error (Compiler) 208
Critical Errors 216
CSeg 19

D
{$D+/-} 181
Datei nicht gefunden 196
Datei nicht gefunden (E/A) 213
Datei nicht geöffnet 215
Datei nicht zugewiesen 215
Datei-Handle ungültig 214
Datei-Zugriff verweigert (Compiler) 210
Datei-Zugriff verweigert (E/A) 214
Dateifehler 215
Dateiname ungültig 196
Dateityp ungültig 202
Datensegment zu groß 200
Dec 19
DEFINE (bedingte Compilierung) 190
{$DEFINE} 190
Delay 20
Delete 20
DelLine 20
Destruktor-Bezeichner erwartet 211
DetectGraph 21
Diskette schreibgeschützt 216
Diskette voll 196
Diskette-Lesefehler 215
Diskette-Schreibfehler 215
DiskFree 23
DiskSize 23
Dispose 23
Division durch Null (Compiler) 202

Division durch Null (Laufzeit) 217
DosExitCode 24
DosVersion 25
DrawPoly 25
DSeg 26

E
{$E+/-} 182
Ein-/Ausgabe-Prüfung 183
Ein-/Ausgabefehler 213
Einbinden von Object-Dateien 187
Ellipse 27
ELSE (bedingte Compilierung) 191
{$ELSE} 191
Emulator (Coprozessor) 182
ENDIF (bedingte Compilierung) 191
ENDIF fehlt 208
{$ENDIF} 191
EnvCount 28
EnvStr 28
Eof 28
Eoln 29
Erase 30
Erweiterte Syntax 186
Erzeugung von 80286-Code 183
Erzwingen von far-Aufrufen 182
Exec 31
Exit 32
Exp 33

F
{$F+/-} 182
Fail 33
Fatal Errors 217
Fehlercodes von DOS 213
FExpand 34
FilePos 34
FileSize 35
FillChar 35
FillEllipse 36
FillPoly 37
FindFirst 38
FindNext 39
FloodFill 40
Flush 41
FOR-Variable ungültig 205
Format-Spezifizierer ungültig 209
Frac 42
FreeMem 42
FSearch 42
FSplit 43

G
{$G+/-} 183
GetArcCoords 45
GetAspectRatio 46
GetBkColor 47
GetCBreak 48
GetColor 48
GetDate 49
GetDefaultPalette 49
GetDir 51
GetDriverName 51
GetEnv 52
GetFAttr 53
GetFillPattern 54
GetFillSettings 55
GetFTime 55
GetGraphMode 56
GetImage 57
GetIntVec 58
GetLineSettings 59
GetMaxColor 60
GetMaxMode 60
GetMaxX 61
GetMaxY 62
GetMem 62
GetModeName 63
GetModeRange 63
GetPalette 64
GetPaletteSize 65
GetPixel 65
GetTextSettings 66
GetTime 67
GetVerify 67
GetViewSettings 67
GetX 68
GetY 69
Gleitkomma-Operation ungültig 218
Gleitkomma-Überlauf 218
Gleitkomma-Unterlauf 218
GotoXY 70
GraphDefaults 70
GraphErrorMsg 71
GraphResult 72

H
Halt 73
Hardware-Fehler 217
Heap-Überlauf 217
Hi 74
HighVideo 74

I
{$I Datei} 186
{$I+/-} 183
IFDEF (bedingte Compilierung) 191
{$IFDEF} 191
IFNDEF (bedingte Compilierung) 191
{$IFNDEF} 191
IFOPT (bedingte Compilierung) 191
{$IFOPT} 191
ImageSize 74
Inc 75
Include-Datei 186
InitGraph 76
INLINE-Fehler 206
Insert 79
InsLine 79
InstallUserDriver 80
InstallUserFont 81
Int 82
Integer-Konstante fehlerhaft 195
Intr 82
IOResult 83

K
Keep 84
KeyPressed 85
Kollektion-Index außerhalb Bereich 219
Kollektion-Überlauf 219
Konstruktor-Bezeichner erwartet 211

L
{$L Datei} 187
{$L+/-} 184
Laufwerk nicht bereit 216
Laufwerksnummer ungültig 215
Laufzeitfehler 213
Length 85
Lesefehler des Geräts 217
Line 86
LineRel 87
LineTo 87
Ln 88
Lo 89
Lokale Symbole 184
LowVideo 89

M
{$M ...} 187
Mark 89
Marke nicht definiert 204
Marken doppelt definiert 204
MaxAvail 90
MemAvail 91
MkDir 91
Move 92
MoveRel 93
MoveTo 94
MsDos 94

N
{$N+/-} 184
New-Prozedur/Funktion 95
NormVideo 96
NoSound 96
Numerische Operationen 184
Numerisches Format ungültig 216

O
{$O Unit} 188
{$O+/-} 184
Objekt nicht initialisiert 219
Odd 96
Ofs 97
Operandentypen nicht passend 199
Ord 97
OutText 97
OutTextXY 99
Overlay-Datei-Lesefehler 218
Overlay-Deklaration 188
Overlay-Prüfung 184
Overlay-Verwaltung nicht installiert 218
OvrClearBuf 100
OvrGetBuf 100
OvrGetRetry 101
OvrInit 101
OvrInitEMS 102
OvrSetBuf 103
OvrSetRetry 103

P
PackTime 104
ParamCount 104
ParamStr 105
Pfad nicht gefunden 213
Pi 106
PieSlice 106
Pos 107
Pred 107
Ptr 108
PutImage 109
PutPixel 110

R
{$R+/-} 185
Random 111
Randomize 111
Read 112
ReadKey 115
Readln 116
Real-Konstante fehlerhaft 195
Rectangle 117
RegisterBGIDriver 118
RegisterBGIFont 120
Release 122
Rename 122
Reset 123
RestoreCrt 124
RestoreCrtMode 125
Rewrite 125
RmDir 126
Round 127
RunError 127

S
{$S+/-} 185
Schreib-/Lesefehler 216
Schreibfehler des Geräts 217
Sector 128
Seek 129
SeekEof 130
SeekEoln 130
Seg 131
SetActivePage 131
SetAllPalette 132
SetAspectRatio 134
SetBkColor 135
SetCBreak 136
SetColor 136
SetDate 137
SetFAttr 137
SetFillPattern 138
SetFillStyle 140
SetFTime 141
SetGraphBufSize 141
SetGraphMode 142
SetIntVec 143
SetLineStyle 144
SetPalette 146
SetRGBPalette 147
SetTextBuf 149
SetTextJustify 151
SetTextStyle 152
SetTime 153
SetUserCharSize 154
SetVerify 155
SetViewPort 155
SetVisualPage 157
SetWriteMode 158
Sin 159
SizeOf 160
Sound 160
Speicherbelegung 187
Speicherüberlauf 194
SPtr 161
Sqr 161
Sqrt 161
SSeg 161
Stack-Prüfung 185
Stack-Überlauf 217
Statement-Fehler 206
Str 162
Stream-Registrationsfehler 219
String-Konstante über Zeilenende 195
Strukturen zu groß 197
Strukturierte Variablen unzulässig 209
Succ 162
Swap 163
SwapVectors 163
Syntaxfehler 195

T
TextBackground 164
TextColor 164
TextHeight 165
TextMode 167
TextWidth 168
Trunc 169
Truncate 169
TypeOf 170
Typumwandlung ungültig 202

U
Überprüfung von var-Strings 185
UNDEF (bedingte Compilierung) 190
{$UNDEF} 190
Ungültiger Zeigeroperator 217
Unit unbekannt 216
Unit-Dateiformat fehlerhaft 203
Unit-Name doppelt 203
Unit-Versionen passen nicht 203
UnpackTime 170
UpCase 171

V
{$V+/-} 185
Val 171

W

Wertebereich (untere/obere Grenze) 198
WhereX 172
WhereY 172
Window 173
Write 174
Writeln 177

X

{$X+/-} 186

Z

Zeigervariable erwartet 202
Zeile zu lang 196
Zu viele offene Dateien 196
Zu viele Variablen 205
Zusatzinformationen zur Fehlersuche 181
Zuweisung unzulässig 199

Gestärkt durch die Leistungsfähigkeit und Effektivität der objektorientierten Programmierung gleibt die Stellung von Turbo Pascal als Marktführer auch in Zukunft gesichert... Sicher ist nur eins, OOP wird neue Bereiche eröffnen, sei es in Datenbank-Management-Systemen oder auch in Programmiersprachen der Zukunft.

Jürgen Mertes/Klaus Schäfer
PC Magazin, 18. Oktober 1989

Das objektorientierte Modell hat das Zeug dazu, nach der Einführung der strukturierten Programmierung vor gut zwei Jahrzehnten die zweite »sanfte« Wende auf dem Gebiet der Software-Entwicklung zu sein.

Ronald Fischer
Toolbox, Oktober 1989

B O R L A N D